SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–40
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

MobSTer: A Model-based Security Testing Framework for Web Applications

Michele Peroli[1], Federico De Meo[1], Luca Viganò[2] and Davide Guardini[1]

[1] *Dipartimento di informatica, Università di Verona, Italy.* {*michele.peroli | demeof | davide.guardini*}*@gmail.com*
[2] *Department of Informatics, King's College London, UK. luca.vigano@kcl.ac.uk*

SUMMARY

Web applications have become one of the preferred means for users to perform a number of crucial and security-sensitive operations such as selling and buying goods or managing bank accounts, official documents, personal health records, smart houses and so on. The pervasive adoption of such web applications calls for an extensive security analysis in order to avoid attacks. Penetration testing is the most common approach for testing the security of web applications, but model-based security testing has been steadily maturing into a viable alternative and/or complementary approach. Penetration testing is very efficient but the experience of the security analyst is crucial; model-based security testing relies on formal methods but the security analyst has to first create a suitable model of the web application. In this paper, we introduce MobSTer, a formal and flexible model-based security testing framework that contributes to filling the gap between these two security testing approaches. The main idea underlying this framework is that the use of model-checking techniques can automate the search for possible vulnerable entry points in the web application, i.e., it permits an analyst to perform security testing without missing important checks. Moreover, the framework also allows for reuse: the analyst can collect her expertise into the framework and (re)use it during future tests on possibly different web applications. We have implemented MobSTer as a prototype and applied it to test a number of case studies to assess its strength and concretely evaluate it with respect to four state-of-the-art tools normally used by penetration testers.

Received . . .

KEY WORDS: Security Testing; Web Applications; Model-Checking; Model-based Testing.

1. INTRODUCTION

1.1. Context and motivation

Different approaches can be followed to assess the security of a web application, such as the formal analysis of the web application's specification or the testing of the web application's code [20]. *Penetration testing* [32, 42] is the most common approach for testing the security of web applications, i.e., for generating a set of test cases that simulate real attacks and executing them on the web application in order to acquire more confidence about the secure behavior of an application's implementation or to discover failures (i.e., unexpected behaviors). *Model-based security testing* [19] (i.e., model-based testing [15, 43] of security requirements) is not yet as widespread as penetration testing but it has been steadily maturing into a viable alternative and/or complementary approach (as discussed by Binder et al. [6], model-based testing has had

[1]Correspondence to: Luca Viganò, Department of Informatics, King's College London, Strand Campus, Bush House, 30 Aldwych, London WC2B 4BG, UK. Phone: +44 020 78482078. E-mail: luca.vigano@kcl.ac.uk

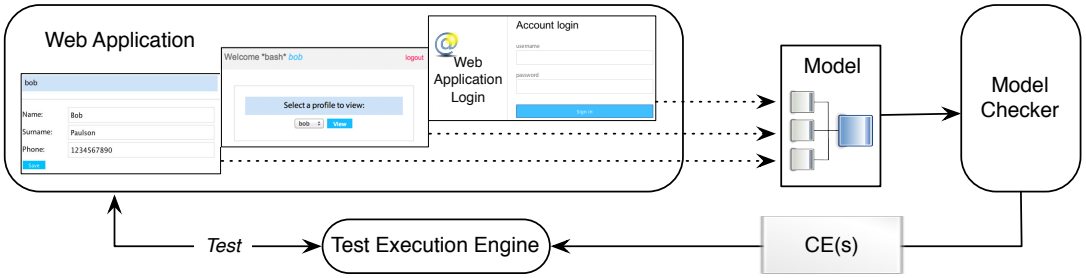

Figure 1. A high-level view of the MobSTer framework.

positive effects on efficiency and effectiveness of the testing methodologies adopted in various organizations). Both these testing techniques, however, require quite some effort of the security analyst carrying out the tests, even when she[2] may make use of existing tools, guidelines or libraries of common security vulnerabilities and attacks such as [13, 14, 29, 34, 32, 39]. In particular: penetration testing, which ranges from *black-box* to *white-box*, has helped uncover several vulnerabilities, but the experience of the security analyst carrying out the pen-tests is crucial for their success, as it is well known (cf. [17]) that the use of automatic tools is not enough for determining the overall security of the web application. In model-based testing, a formal model of the web application is used to formally derive test cases, but this requires the analyst to first create such a model, which may be a difficult endeavor especially in the industrial setting.

1.2. Contributions

In this paper, we introduce *MobSTer*, a formal and flexible *Model-based Security Testing Framework* that supports a security analyst in carrying out security testing of web applications.

The main idea underlying MobSTer is a hybrid approach that takes advantage of model-checking techniques combined with the knowledge provided by penetration testing guidelines and checklists. This combination enables MobSTer to exploit the automatic search for possible vulnerable "entry points" without missing important checks. MobSTer also allows for reuse: the analyst can collect her expertise into the framework and (re)use it during future tests on possibly different web applications.

More specifically, as shown in Figure 1, MobSTer follows the classical approaches for model-based testing. First, the security analyst creates a *model* of the web application by defining *actions*; an action, intuitively, is a part of the web application providing some particular functionality that can be accessed by users through a user interface or a web browser (e.g., authentication, user profile management or item purchase in an e-shopping application). The model is enriched with a security goal that the web application should respect during its execution, and the enriched model is then fed into a *Model Checker* that will return *Counterexamples (CEs)* if any are found, which are execution traces that violate the security goal. However, both the actions and the counterexamples are too abstract to be directly employed for testing the web application. MobSTer thus provides for a *Test Execution Engine (TEE)* that translates a counterexample into a sequence of *HTTP requests* that can be performed on the web application.[3]

As will become evident below, actions are simple to identify and can be easily reused. Hence, the presence of actions in our model-based testing framework brings along a number of advantages, including

- *simplicity* (e.g., of the modeling activity),

[2]Throughout the paper we will refer to the security analyst as "she".
[3]The formalization used in our approach can also be adapted to represent software that is not web-based. However, our concretization phase (which is important for testing the results of the formal phase) works specifically for web-based software. Thus, migrating the approach to non-web-based software would require one to address different challenges related to the low-level behavior of such software.

DOI: 10.1002/stvr

- *reusability* (in our framework, actions are associated with data they handle and properties they have, thus, reusing actions also means to reuse these associations during the testing of different web applications).

Actions are defined by the security analyst who decides the abstraction level and the granularity she wishes to consider. As a concrete example of the actions' characteristics (i.e., their use in the framework as well as the expertise used in their identification), consider the "login" functionality. Web applications usually employ one of the following types of authentication mechanism (we also add the attacks that the security analyst can exploit in order to attack the action):

- *Basic Authentication*: the tester can try to perform a brute-force attack, and the password is sniffable if not passed via HTTPS.
- *HTTP Digest Authentication*: brute-force attacks and man-in-the-middle attacks are possible.
- *Form-based Authentication*: brute-force attacks are possible, injection vulnerabilities (e.g., SQL-injection [38], Cross-Site Scripting XSS [22], etc.) have to be tested, and the attacker should try to capture the authentication token (i.e., session tokens or cookies).

For each of these authentication types, a security analyst can create an associated action and also reuse it later to test different models. In fact, this simple example shows how, for a security analyst modeling the web application, it should be *simple* to identify an action for the different web application functionalities (e.g., the authentication mechanism), and to *scale* or *reuse* some existing action (e.g., she can modify/extend a "form-based authentication" action if the web application uses some particular data or has different properties, or she can take an action for authentication from the database). For concreteness, in the examples and the case study considered later, the form-based authentication will be used.

Furthermore, the use of model-checking techniques allows us to reason about logic flaws related to the behavior of the web application. Logic flaws of web applications are an important class of "defects" that are the result of faulty application logic but remain outside the scope of most of existing tools since they require an exhaustive understanding of the workflow and dataflow of the web application. Thanks to the formalization method proposed in this paper, it is also possible to detect and exploit logic flaws in web applications covering a wide range of attacks that no other tool is able to cover.

This paper shows the range of tests that can be performed with MobSTer rather than focusing on the performances of the tests. To do so, we have implemented MobSTer as a prototype[4] and applied it to test a number of case studies to assess its strength and concretely evaluate it with respect to four free, non-commercial, state-of-the-art tools that are normally used by penetration testers and that are close to what MobSTer aims to achieve: Burp Suite [36] (free version 1.7.23 and Pro version 1.7.23), OWASP Zed Attack Proxy [31] (ZAP, version 2.3.1), Paros [12] (version 3.2.13) and Arachni [2] (framework version 1.5.1). Our evaluation shows that MobSTer has a better identification rate and a better vulnerability coverage than these four tools.

1.3. Organization

In Section 2, we describe MobSTer. In Section 3, we briefly introduce some basic notions of the Alloy language and we show how the model of a web application (as described in Section 2) is written in Alloy. In Section 4, a first case study is used to show the framework at work on two examples, highlighting how actions can be used to give an abstraction of the web application. In Section 5, we give details of the tests on other case studies along with details of the implementation of MobSTer using the Python language. In Section 6, we discuss related work. In Section 7, we draw conclusions and discuss future work.

[4]The source code of the MobSTer tool is available at `https://github.com/REGIS-lab/MobSTer`.

DOI: 10.1002/stvr

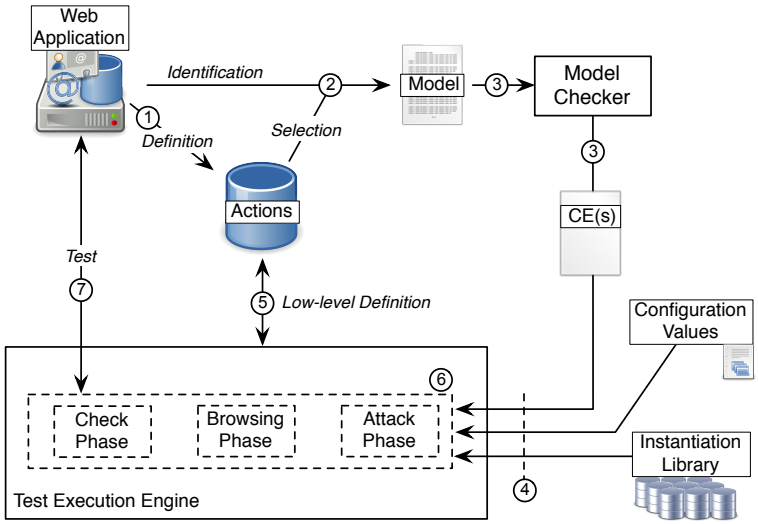

Figure 2. MobSTer, a framework for model-based testing of web applications (the arrows refer to interaction between elements or data passed between them, the numbers are used in the text for the explanation of the different phases).

2. A FRAMEWORK FOR MODEL-BASED SECURITY TESTING OF WEB APPLICATIONS

Our MobSTer framework is depicted in Figure 2, which refines Figure 1 and is described in detail in Section 2.1. MobSTer uses the actions of the web application both in the modeling of the web application itself (Section 2.2) and in the concretization of the test cases (Section 2.3). Intuitively, we define an action as an abstract representation of a functionality provided by the web application that can be "used" through a user interface (e.g., a web browser), where *using* an action means that it is possible to perform a sequence of ordered HTTP requests leading the user to access a functionality provided by the action.

Web applications offer various types of functionalities, ranging from general-purpose functionalities such as authentication, editing of private information or searching information, to specific functionalities such as reading a newsfeed or purchasing goods from an online shop. During the modeling phase, the implementation details of a single action are not necessary to consume it. The modeling focus is on the interaction between actions themselves and between actions and the data they have access to. For example, the authentication action requires credentials in order to authenticate a user and we specify that administrative actions, such as modifying the personal profile, can be performed only after the action for authentication (i.e., the update profile action comes after the login action).

2.1. *The phases of our framework*

The first thing that a security analyst has to do when using MobSTer is to define actions from the web application (phase ① in Figure 2). During this phase, the security analyst has also to check, manually, if some of the existing actions in the database can be reused to model the web application (if two web applications share the same functionality, then the analyst can reuse the associated action), and, if not, she has to insert into the database the new action(s); see Section 2.2 for further information about the definition of actions.

With the database populated with a proper set of actions, the security analyst has the means to create the *model* of the web application (②). The model includes a subset of the defined actions (identified with respect to the web application), the relation between them, and a specification of the security goal to be tested.

The model is then passed to a *model checker* (③). Our framework is general and thus it is not bound to a specific model-checking tool; for concreteness, the implementation of MobSTer employs

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the *Alloy Analyzer* [24, 25], which takes a model and its security goal, checks the goal in the model and generates one or more counterexamples if the goal is violated, i.e., a counterexample shows for what instances of the system and for what actions the security goal does not hold.

The fact that the counterexamples are abstract gives, however, rise to two problems (related to the level of abstraction) that have to be tackled. First, the counterexample(s) are at a level of abstraction that does not permit one to directly test them on the web application, since it specifies the actions used to violate the goal but not how these actions should be used in the real implementation. MobSTer thus provides for a *concretization phase*, which relies on the fact that for each action it is possible to define a sequence of HTTP requests to perform on the web application. In the implementation of MobSTer, the definition of the relations between actions and HTTP requests (⑤) is performed during the execution of the test cases by a Python engine.

Second, the counterexample(s) specify the actions to use, but their level of abstraction does not allow for the specification of attack-dependent data. The *Instantiation Library* provides specific payload to use when needed. It contains data such as attack strings (e.g., payloads for XSS), common malicious input (e.g., a set of passwords for a brute-force attack) and scripts to be used as test patterns (i.e., scripts to be executed client-side in order to test the web application).

The final phase of the framework uses an automatic test-execution engine TEE that the security analyst can employ to execute the test cases on the web application. The TEE provides a connection with the Instantiation Library and the data that is contained in the *Configuration Values* and is needed for the interaction with the web application (④). The TEE also takes care of "translating" (via the *Low-level Definition* ⑤) the counterexample(s) into executable test cases (④). At the end of this phase, the information contained in the counterexample(s), the actions and the HTTP requests are combined in the creation of a suite of test cases (⑥) that are run on the web application (⑦).

2.2. Modeling web applications for security testing

2.2.1. Defining the transition system states In MobSTer, the definition of models is inspired by the *HRU model*, a security model proposed by Harrison, Ruzzo and Ullman for the integrity of access rights in operating systems [23]. The HRU model defines (i) a finite set of generic rights R and (ii) a finite set C of commands containing conditions to be checked and primitive operations to be performed (i.e., create/destroy objects/subjects, give/delete rights on an object). A *configuration* of such a protection system is a triple (S, O, P), where S is the set of current subjects, O is the set of current objects, $S \subseteq O$, and P is an access matrix, with a row for every subject in S and a column for every object in O.[5] $P[s, o]$ is a subset of R, the generic rights. $P[s, o]$ gives the rights to object o possessed by subject s. The "safety" problem for protection systems under this model is to determine in a given situation whether a subject can acquire a particular right to an object. Basically, safety means that an unreliable subject cannot pass a right to someone who did not already have it (i.e., the owner gives away certain rights to his objects).

To test web application using MobSTer, an access matrix M like the matrix P of the HRU model is defined, where the commands are instantiated with the functionalities offered by the web application, the set of generic rights is redefined to express the information that permits one to relate web application functionalities to known vulnerabilities, and the set of primitive operations is changed according to the changes performed on the other concepts.

In MobSTer, the model of a web application for security testing is defined as a *transition system* TS that contains: (i) a data structure containing the data handled by the web application (and its users), (ii) the information about the storage and the management of this data by the web application, (iii) the functionalities that the web application provides, and (iv) how these functionalities can be accessed by the web application's users.

[5] An access matrix can be envisioned as a rectangular array of cells, with one row per subject and one column per object. The entry in a cell (i.e., the entry for a particular subject-object pair) indicates the access mode that the subject is permitted to exercise on the object. Each column is equivalent to an access control list for the object; and each row is equivalent to an access profile for the subject.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Users, data and knowledge The set $UserName$ is defined as the set of unique identifiers of users interacting with the web application, where a special label $Anon$ is introduced for anonymous browsing. For the users in $UserName$, the security analyst has to establish which data is in the scope of the analysis.

Definition 1 ($UserData$)
Let $MetaData$ be a set of abstract representations (defined by the security analyst) of the data implemented in web applications that a user can handle, $|MetaData| = n$ be the number of elements in $MetaData$, $BasicTypes = \{String, Int, Bool, \ldots\}$ be the set of concrete data types and $StrucTypes = \{Profile, Credential, \ldots\}$ the abstract types of the elements of $MetaData$. The record $UserData = (field_1, ..., field_n)$, where $field_i \in UserData$[6] is an instantiation (at some level of abstraction) of the i-th element of $MetaData$ and is of the form $field_i = [(subfield_{i.1}[, subfield_{i.2}], ...)]$, where

- *$field_i$ has type in $StrucTypes$ or $BasicTypes$, and*
- *$subfield_{i.j}$ are optional and have types in $BasicTypes$.*

As an example, a typical data structure is

$$\begin{aligned} UserData = & (cred = (user, pwd), \\ & id, \\ & prof = (name, ...), \\ & messages = (m[, m]...[, m]), \\ & data = (d[, d]...[, d])). \end{aligned}$$

When a security analyst models a web application, some of the fields of $UserData$ will remain unchanged, others will be modified (the choice will depend on the web application in some cases and on the modeling choices in other cases).

In a multi-user environment, every user has access to his (and other users') data through the interface of the web application. To model this aspect, the set $Data$ contains the possible data that users can handle during their interaction with the web application.

Definition 2 ($Data$)
The set $Data$ is defined by instantiating every element in $UserData$ with each user in $UserName$, i.e.,

$$Data = \{x.y \mid x \in UserName \text{ and } y \in UserData\}.$$

During the interaction with a web application, a user can access and use many types of knowledge. For instance, he can:

- use information he already knows, e.g., from the beginning of his execution (the origin of this initial knowledge is not discussed here and its definition is demanded to the security analyst),
- gain information from the interaction with the web application itself,
- or, in borderline cases, even guess some information.

These types of knowledge are described by means of labels contained in the set

$$K_{Source} = \{Initial, Gained, Guessed\},$$

where other sources of knowledge can of course be modeled by defining a different K_{Source}.

Definition 3 (Knowledge)
The knowledge of the users is defined as a set of triplets

$$U_{Knows} = \{(x, d, k_{src}) \mid x \in UserName, d \in Data \text{ and } k_{src} \in K_{Source}\},$$

and $U^{s_i}_{Knows}$ denotes the content of the set U_{Knows} at state s_i.

[6] With a slight abuse of notation, we use $\in$ also for records.

DOI: 10.1002/stvr

The behavior of web applications The behavior of a web application is modeled through events that a specific user triggers (i.e., causes to happen), through the use of the functionalities of the web application (i.e., actions), regarding some data, and with respect to a specific "location" on the server. In MobSTer, events describe what is happening to the web application's data. Events are related to actions in that when an action α is performed, some events take place (i.e., the user triggers some events through the use of the action).

Definition 4 (Events)
Events describe how the data is managed by the information technology that the web application relies on (e.g., databases that manage data, file systems for files, sessions for access control and volatile data, operating systems that execute commands, and the web application's user interface that retrieves data from the users). The syntax for defining events *is*

$$x.event(parameters, location)\,,$$

where $x \in UserName$, $event$ *is the event's name,* $parameters \subseteq Data$ *and* $location$ *is the technology used by the web application to handle the data (e.g., the file system).*

For instance, if an action models a functionality that allows a user to write some data on the web application's database, then the event $x.write(targetData, database)$ is related to that action.

As we will show in the following sections, actions (Definition 9) are used to move the transition system from a state (Definition 8) to another. In order to simplify the notation for the actions, events are specified in the target state of the action, i.e., for states s_i and s_{i+1} of the transition system,

$$s_i \xrightarrow{\alpha_i+\{events\}} s_{i+1} \quad \text{becomes} \quad s_i \xrightarrow{\alpha_{i+1}} s'_{i+1},$$

where in s'_{i+1} for each event in $\{events\}$, a label expressing the $event$ and its $location$ is specified in the event's $parameters$. In those cases in which it is not possible to determine the location where the data are managed, the security analyst can create multiple models with different guesses, and test each of the resulting models. The automated test generation and execution process will take care of reducing the overhead of testing multiple traces.

Similar to the definition of the knowledge of the users, the set $Event$ contains the possible labels for events. As a concrete example, a useful instantiation of the set $Event$ (that could, of course, also be modified by the security analyst) is

$$Event = \{ShowDB, WriteDB, ShowFS, WriteFS, Exec, Edit, WriteSD, ShowSD\}\,,$$

where the intended meanings of the labels in $Event$ are:

- $ShowDB$: the labeled data has been displayed as a result of a query that reads from a database (e.g., the messages in an online forum).
- $WriteDB$: the labeled data has been written in a database (e.g., if a user saves his profile on a database).
- $ShowFS$: the labeled data has been read from the file system and displayed (e.g., the web application has a photo album whose photos are read from files).
- $WriteFS$: the labeled data has been written on the file system of the server (e.g., photos, attached documents, etc.).
- $Exec$: the labeled data has been displayed and retrieved as part of the execution of a command (e.g., the open function in PERL or the *Runtime class* in Java).
- $Edit$: the labeled data has been retrieved by the web application for editing (e.g., a form that a user can edit).
- $ShowSD$: the labeled data has been retrieved and displayed from a local session of the browser (e.g., preferences or runtime state).
- $WriteSD$: the labeled data has been saved in the browser along with the other data pertaining to a certain session.

DOI: 10.1002/stvr

There is a slight difference between a $ShowDB$ (or the other types of "$Show$") and an $Edit$, namely, $ShowDB$ refers to the fact that the data is only displayed, $Edit$ to the fact that a user can change the values of the data. The action $Edit$ is used in those cases in which the web application permits a user to modify the values of the data (e.g., an HTML form, where the data can be modified, is displayed to the user), whereas $ShowDB$ is used when the web application only displays the data without providing means for modifying it.

Definition 5 (WA_{Event})
The set of events that are related to the user and the data is defined as:

$$WA_{Event} = \{(x, d, e) \mid \ x \in UserName, d \in Data \text{ and } e \in Event\},$$

where a triplet in WA_{Event} states that the event e happens on a data d and the user x triggered it. $WA^{s_i}_{Event}$ denotes the content of WA_{Event} at state s_i.

Security Mechanisms & Testing-Related Information Users interact with a web application through a browser. Even if a user is not aware of what is happening, from a security perspective, a lot of information can be extracted from a web application about the mechanisms (that are often concealed to the users) implemented in order to preserve the security of the system (i.e., the security of the web application and its data-management technologies), and the information that is interesting from a testing perspective but is not part of the knowledge or the behavior of the web application.

Definition 6 (Assertions)
Assertions describe security controls such as authentication, access control, input validation, encoding, and user and session management. A security analyst can define assertions (in the form of labels) about security controls in order to define the set $Assertion$.

The analysis is focused on those security mechanisms (enforced through/on some data) that refer to the classes in Definition 6. The strategy that a security analyst can follow when defining the set $Assertion$ is to identify the key attack surfaces that web applications can expose. This strategy corresponds to mapping the attack surface of the web application; some key areas to investigate during the mapping are:

- the web application's functionalities (i.e., the actions that can be leveraged);
- the core security mechanisms (e.g., access controls, authentication mechanisms, etc.);
- how the web application processes user-supplied input;
- the technologies employed on the client side;
- the technologies employed on the server side.

As an example, the following set is the one used in the case study:

$$Assertion = \{Granted, Checked, AJAX, Sanitized, Admin, User, Echoed, PageIncluded, noAttack\},$$

where the intended meanings for the elements in $Assertion$ are:

- $Granted$: this label is used along with the data modeling the (HTTP) session and means that the session is granted (i.e., the user has logged-in and some session ID is used during the communication).
- $Checked$: if the data is used in a query on the database (or file system) and may not be displayed by the user interface.
- $Sanitized$: if sanitization is enforced on the data.[7]

[7]Since we are performing model-based testing, the only granularity we consider is fully sanitized or not. The concretization phase will take care of testing the generated traces.

DOI: 10.1002/stvr

- $Admin/User$: if the role of the users of the web application is checked (these assertions define the values of the user data *userType*, and are checked whenever an action requires these privileges).
- $AJAX$: if the data was displayed and its values are retrieved via AJAX-functionalities.
- $Echoed$: if the data submitted through a request is reported identical in the response page.
- $PageIncluded$: if in the URL/page there is a direct reference to a file (then used as a web page) hosted on the server.[8]
- $noAttack$: this label is used as a means for a security analyst to disable attacks for a certain data.

Definition 7 ($SEC_{Assertion}$)
The set

$$SEC_{Assertion} = \{\ (x, d, p)\ \mid\ x \in UserName, d \in Data \text{ and } p \in Assertion\ \}$$

states that a certain user x has used a data d on which the security analyst has made an assumption p about how the data d is handled from a security perspective; $SEC^{s_i}_{Assertion}$ denotes the content of the set $SEC_{Assertion}$ at state s_i.

States of the transition system In our framework, every state s_i describes a particular snapshot of the web application regarding the information about: (i) the users in $UserName$ and their knowledge in $U^{s_i}_{Knows}$, (ii) the triggered events in $Event$ on the data ($WA^{s_i}_{Event}$), and (iii) the assertions in $Assertion$ about security mechanisms and testing-related information $SEC^{s_i}_{Assertion}$. In other words, a state describes what is happening client-side and server-side during the interaction of a user with the web application.

Definition 8 (States)
For a given web application, a state *of the TS is an instance of the matrix M such that the row names take values in $UserName$, the column names take values in every element of $Data$ and the labels in K_{Source}, $Event$ and $Assertion$ are assigned to M's cells. The syntax $M[U, D]$ denotes a cell of the matrix M of the states of TS, where $U \in UserName$ and $D \in Data$.*

A final remark on the matrix is in order. As stated above, our approach takes inspiration from the *HRU model* [23]. The definition of the access matrix (P in the original paper) remains the same but, in our approach, "subjects" are replaced by "users" and "objects" are replaced by "data" (the definitions of "commands" and "rights" are replaced in order to be usable in the context of web applications).

2.2.2. Actions Let the following set be a set of labels for the modeled functionalities

$$\begin{aligned} functionName = \{&Login, Logout, Search, GetEdit, ListId,\\ &EditProfile, ViewProfile, UpdateProfile\}. \end{aligned}$$

The elements in $functionName$ refer to the functionalities implemented in the web application that are modeled as actions (in Figure 2 this step is labeled as "identification phase"). These actions have to be instantiated with respect to the data (contained in the set $Data$) of the web application.

Definition 9 (Actions)
An action $\alpha \in Action$ *is defined as*

$$\alpha = name(agent, parameters)/[Conditions]PrimitiveTransitions\,,$$

[8] Above we have introduced the event $ShowFS$ that could be seen as a repetition of $PageIncluded$. The two concepts are indeed similar but they differ in the fact that we see the event $ShowFS$ as some data that is parsed to be included in a page, whereas the assertion $PageIncluded$ refers to the actual files where web pages are saved (e.g., files with extension html, php, or asp).

DOI: 10.1002/stvr

Table I. Definition of the *Login* action.

```
1 Login(x, x.cred)
2  if (M[Anon, Anon.session] = Grant ∧ M[x, x.cred] = Initial)
3    Reset M for x
4    Del Grant from M[Anon, Anon.session]
5    Add Grant into M[x, x.session]
6    Add Checked into M[x, x.cred]
7 End
```

where $name \in functionName$, $agent \in UserName$, $parameters \subseteq Data$, $Conditions$ *is a set of conditions that have to be satisfied in order to perform the action, and* $PrimitiveTransitions$ *is a set of transitions that describe how the state changes.*

The elements of $PrimitiveTransitions$ are of the form

$$\texttt{operation } X \texttt{ [into/from] } M[U, D]$$
$$\texttt{operation } X \texttt{ for } U$$

where $X \in Event \cup Assertion$, $U \in UserName$ and $D \in Data$. In the above example, the first primitive transition is applied to a single cell of M, whereas the second one is applied to all the cells in the row U. For instance, the following primitive transitions are used for the definition of actions in our case study:

- `Add` X `into` $M[U, D]$ — X is appended in the cell $M[U, D]$,
- `Del` X `from` $M[U, D]$ — X is deleted from the cell $M[U, D]$, and
- `Reset` M `for` U — in the row U of M every value that differs from *Initial* or *Gained* or *Granted* is deleted.

In the last transition, the values of the knowledge are not deleted in order to maintain a monotonic knowledge of the users; we also assume that the *Granted* assertion can be deleted only with the application of an action.

A *condition* is an expression of the form:

$$\texttt{if } X \quad \in \texttt{ / } \notin \quad M[U, D]\,.$$

As an example, consider the definition of the action *Login*, which is given in Table I. The action's *name* is "Login" (line `1`) and, to maintain the functionality general enough to be used by multiple agents, let x be the agent using it. As *parameters* both "username, password" and "credential" could be used (instantiated for the user x). The parameter "credential" (*cred* for short) is used in the example in Table I.

Usually, a login can be performed only if the user is not logged in yet (i.e., he is still anonymous to the web application) and if he knows his credentials (i.e., the credentials are part of his knowledge). These conditions are defined in line `2` of Table I. Once the conditions in line `2` have been fulfilled, the state of the transition system needs to be changed. First of all, the previous event is deleted from the matrix (through the primitive `Reset` in line `3`), then the *Grant* label is deleted from the *Anon* user in line `4` (this also means that a user can login only from an anonymous session) and the user receives the session in line `5`. This information is also stored in the state (line `6`) since the "credentials" are checked (the low-level mechanism is not important) and the action is closed in line `7`.

2.2.3. Defining security goals The purpose of security goals is to verify that some properties or conditions hold on the model. In this paper, we restrict the attention to some security goals that represent well-known vulnerabilities (like the ones discussed in Section 5), but it is important to stress that MobSTer is open to considering more complex vulnerabilities.

In general, to exploit a vulnerability, some conditions related to the data and some properties must be fulfilled. Conditions are formalized with respect to a vulnerability by means of a logical formula that has to be valid in every possible state describing the evolution of the model, or has to be valid for every trace starting from an initial state.

DOI: 10.1002/stvr

Although it is not possible to give an exact procedure for the definition of security goals, in the following, we give a general approach that can be helpful to a security analyst while writing security goals. The approach consists of four steps, which are described below. Each step should be instantiated according to the vulnerability for which the security analyst wants to write the security goal; examples of instantiations are given after each step.

1. *Definition of entry points:* The security analyst has to determine which are the entry points that are used to attack a web application. Examples of entry points are:

   - text/numerical parameters,
   - URLs,
   - login functionalities and
   - paths to files.

2. *Understanding the testing procedure:* The security analyst then has to understand which is the procedure used to test the vulnerability for which she wants to define a security goal (usually this procedure is the same as the one that is used to attack the web application). The OWASP testing guide [34] has an "How to Test" section for each covered vulnerability, and it is a good starting point for learning how to test web applications. For instance, the procedure for testing "SQL-Injections" described in [34] is:

   (i) Make a list of all input fields whose values could be used in crafting a SQL query.
   (ii) Test them separately, trying to interfere with the query and to generate an error.
   (iii) Monitor all the responses from the web server and have a look at the HTML/JavaScript source code for evidence of a successful attack.

   The procedure for testing "Directory traversal/file include" described in [34] is:

   (i) Enumerate all parts of the application that accept content from the user in order to load static information from a file.
   (ii) Insert malicious strings in the used parameter to include files that are not intended to be accessed by the user (e.g., "../../../../etc/passwd" to include the password hash file of a Linux/UNIX system). It is also possible to include files and scripts located on an external web site.
   (iii) Monitor all the responses from the web server and have a look at the HTML/JavaScript source code for evidence of a successful attack.

   The security analyst will gain two valuable items of information from the testing procedure. First, how the test should be performed, which will be included in the model of the web application as a goal defining how to find the entry point. Second, the set of payloads that should be used during the concretization phase.

3. *Model behavior association:* The security analyst has to define which behavior of the model is more suitable to describe the entry points and the state in which a web application should be after a successful test for the vulnerability. For example, the information that can be used for the definition of a login bypass via "SQL-Injection" is:

   (i) Username and password are submitted to the backend server.
   (ii) Username and password are checked against the information stored in the database.
   (iii) The restricted functionalities of the web application are usable.

4. *Logical formula definition:* The security analyst has to define a logical formula that merges all the information gathered during the previous steps with respect to the sets K_{Source}, $Event$ and $Assertion$ used to model the web application.

In the following, we give some detailed examples of how security goals can be defined in our framework for various vulnerabilities.

DOI: 10.1002/stvr

Initial knowledge and reachable states From a modeling perspective, the labels in K_{Source} are used to give the security analyst enough flexibility during the definition of security goals. As an example, a security analyst can use the initial knowledge to derive those states of the system that can be reachable only through the knowledge of some data. Let's assume that a web application requires the use of a data d_1 in order to access some other data d_2, and that the latter must remain private to each user. In such cases, the security analyst could write a goal to test if, for two users x and y, giving to y the knowledge of x's d_1 could allow y to access x's private data d_2.

Definition 10 (PrivateData – specific scenario)
Let $x, y \in UserName$, $d_1, d_2 \in Data$, $i \in \mathbb{N}$, the matrix M^0 be the initial state, and M^i be the matrix after the use of i actions. In the scenario described above, the private data may be accessed on the web application by a user that does not own it if there is a state M^i at the end of a sequence of states defined with the application of actions (i.e., $M^0\alpha_1 M^1\alpha_2 \dots \alpha_i M^i \alpha_{i+1} M^{i+1} \dots M^{j-1}\alpha_j M^j$) such that $Initial \in M^0[y, x.d_1] \wedge ShowDB \in M^i[y, x.d_2]$.

Such goal aims to test the cases in which the knowledge of some data (acquired by an attacker or security analyst by, e.g., social engineering) could result in privilege escalation.

Stored XSS A *stored XSS* [22] may occur when the attacker saves some data (usually a malicious client-side script) on the server, and then this data is returned to other users in the course of regular browsing. Assuming $WriteDB, ShowDB \in Event$, the goal for a stored XSS corresponds to searching in the model the possible entry points of the web application.

Definition 11 (Stored XSS)
Let $x, y \in UserName$, $d \in Data$, $i, j \in \mathbb{N}$ with $i < j$, the matrix M^0 be the initial state, and M^i and M^j be the matrices after the use of i and j actions. A two-users Stored XSS *may occur on the web application if there is a state M^i at the end of a sequence of states defined with the application of actions (i.e., $M^0\alpha_1 M^1\alpha_2 \dots \alpha_i M^i \alpha_{i+1} M^{i+1} \dots M^{j-1}\alpha_j M^j$) such that $WriteDB \in M^i[x, x.d] \wedge ShowDB \in M^j[y, x.d]$.*

The goal for stored XSS

(i) assumes that users can be dishonest (the same assumption is normally made in penetration testing) and thus an attacker is not considered for the definition of the goal (as normally happens in, e.g., security protocol analysis or approaches that use similar techniques/tools);
(ii) is not bound to a specific web application, since it only depends on the set $Event$, or specific data.

Along with the set $Event$, also the sets K_{Source} and $Assertion$ can be used during the definition of goals; as an example, a security analyst could use the label $noAttack$ in the model, and with a check on the data, discard every trace not adherent with the new goal containing $noAttack$.

Since a goal is a logical formula, if the security analyst reuses the sets K_{Source}, $Event$ and $Assertion$ for the definition of multiple models, then the goal can be reused in the new models. The security analyst can thus create a set of security goals to be reused during her tests without the need of rewriting them every time.

2.3. A concretization methodology: from high to low level

Section 2.2 showed how actions can be used to model a web application; this section shows how actions can be bound to HTTP requests through a concretization methodology that uses a Python engine in order to build HTTP requests. As depicted in the steps ④, ⑤ and ⑥ in Figure 2, the proposed methodology makes use of: the abstract data contained in the counterexample(s) (resulting from the model-checking phase), the Configuration Values needed for the correct interaction with the web application, and the Instantiation Library containing the attack-related information to perform the tests (i.e., payloads and scripts used during the actual attack against the web application).

DOI: 10.1002/stvr

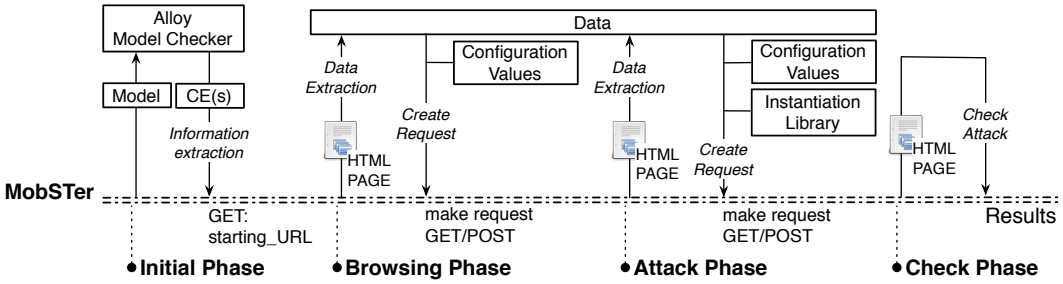

Figure 3. Execution workflow of the MobSTer framework.

Table II. A snippet of a configuration file used by the MobSTer tool.

```
1   starting_URL = 'http://192.168.1.42:8080/WebGoat/...'
2
3   set_cookie = {'JSESSIONID':'974AE36BC95C4I0D036C2E3CDS543B1C'}
4   SSL_authentication = [None|Yes]
5
6   views = {'Login':'ListId'}
7
8   Data['Tom'] = {
9    'employee_id' : '105',
10   'password'    : 'tom'
11  }
12  Data['Jerry'] = {
13   'employee_id' : '106',
14   'password'    : 'jerry'
15  }
16
17  actionsURL = {
18    'NoAction'  : '',
19    'Login'     : 'Login',
20    'Logout'    : 'Logout',
..      [...]
26    'GetSearch' : 'SearchStaff',
27    'Search'    : 'FindProfile'
28  }
29
30  actionsLabel = {
31    'NoAction'  : None,
32    'Login'     : None,
33    'Logout'    : None,
..      [...]
38    'GetSearch' : None,
39    'Search'    : None
50  }
```

As will become evident in Section 4, during the model-checking phase the traces that could be obtained pertain only to honest users of the system. During the concretization phase, instead of assuming that all the users are honest, we assume that users are dishonest; by doing so, honest traces are transformed into attack traces. The reason for doing that is, following the idea of a checklist, to ensure not to miss any possible trace that could bring to an attack.

MobSTer is implemented in Python as an initial proof of concept. The execution workflow of the tool is shown in Figure 3 and is discussed in detail in the following.

2.3.1. Initial Phase In this phase, MobSTer automatically runs the Alloy Analyzer on the model of a web application. If a counterexample is found during the initial phase, it is saved in a text file containing all the information about the states of the transition system. The text file is then

DOI: 10.1002/stvr

parsed, along with the configuration file (containing the Configuration Values), in order to populate the Python variables that will be used in the next phases. Alloy supports multiple counterexamples generations, thus different test cases for the same security property on one model can be generated.

With the information regarding the counterexamples and the relative variables populated, the engine starts the interaction with the target web application. The security analyst is required to write a configuration file (Table II[9]) for the target web application containing: (i) the URL from which to start the interaction (line `1` in Table II), (ii) two optional fields for setting a cookie and stating if the web application requires an SSL connection (lines `3-4` in Table II), and (iii) the list of concatenated actions (line `6` in Table II).

The notion of *views* (i.e., how the actions correspond to the real implementation of the web application with respect to its pages) is introduced in order to define the relations between the "pages" composing a web application and the functionalities provided by it (i.e., how they match). The definition of the views allows the security analyst to model each functionality as a single action without the need of defining actions referring to multiple functionalities. As discussed in the case study presented in Section 4, the functionalities *Login* and *ListId* can be defined for WebGoat; in the actual implementation, after a successful login, the page shown to the user (and thus to the security analyst) contains the result of the concatenated execution of both actions, i.e., the security analyst has a *concatenated view* of the two actions. The Python engine is aware of these modeling choices via the use of the variable `views`.

2.3.2. Browsing Phase Having all the information needed for the interaction with the web application, this phase deals with the problem of reaching the exact location where the attack has to be made. Knowing from the counterexample at which state the attack has to be made (say, at state `5`), the Python engine selects, one by one, the actions from the first state to the one before the attack (in this example, state `4`). For each action, the engine performs two sub-phases: "Data extraction" and "Request creation".

Data extraction: This phase assumes that a web page has been previously retrieved by the engine; for the very first interaction (i.e., the action `NoAction`) the engine retrieves the page pointed by the `starting_URL` variable). The engine extracts and saves the data contained in the retrieved HTML page (e.g., text in input forms, checkboxes, etc.). For each user of the target web application, the data pertaining to the user is maintained in a data structure containing all the data gathered during the analysis (lines `8-15` in Table II).

Request creation: In order to create requests, the Python engine has to retrieve the data pertaining the request and generate the actual HTTP requests that are performed. The following sub-phases are used:

- *Data selection*: The engine checks all the data used in the model (i.e., the data type used during the modeling phase) in order to determine which one should be used (e.g., it tries to discriminate which `employee_id` and `password` should be used for the *Login* action); if multiple data can be selected in order to use an action, the engine automatically checks which data has to be used with a simple comparison with the data displayed, or used, in the subsequent state of the transition system, i.e., by tracking the evolution of the atomic propositions though the different states of the transition system (an example can be found in Section 4.3.1).
- *Input filling*: The engine selects the proper data from the one available in its data structure with respect to the possible input on the page (i.e., the possible HTTP requests that can be made). This is done, at first, by checking (i) if any of the labels (contained in a predefined list) is used in the page (e.g., *text, TEXT, password, textarea*), (ii) a list of indicators for buttons (e.g., *action, submit, button*) and (iii) which information is available in the extracted data (i.e., the data structure `Data`). In case multiple forms are present in a page, this information is not enough to decide which one should be used to create a HTTP request. To resolve this

[9]The example in Table II is also used for the concretization of the case study in Section 4.

DOI: 10.1002/stvr

decisional problem, the security analyst specifies in the configuration file some information (associated with the action) that can be used for making such choices (e.g., the button action or the target page of a link; lines `17-28` in Table II). If the tool is not able to derive any of these information, the missing fields are reported to the security analyst with the request of supplying the missing value.

- *Request generation*: Once the correct data has been selected regarding a certain state of the transition system, the engine is ready to create and send a HTTP request to the web application and, subsequently, to retrieve the resulting web page. The Python engine can generate various types of requests (i.e., GET/POST requests with variables, cookies or SSL authentication). To differentiate such cases, the security analyst should specify for each action a label (referring to the main feature modeled by the action) in the variable `actionsLabel` of the configuration values. The possible values (and their meaning) for `actionsLabel` are:
  - `none`: for buttons or forms (we assume forms to be one of the main features of web applications that are modeled);
  - `link`: for the cases where a link has to be followed in order to reach a different part of the web application (e.g., to reload the information on a web page);
  - `GP`: when the engine has to retrieve the given web page before accessing the functionality (`GP` stands for "Get Page").

2.3.3. Attack Phase Once the browsing phase has been completed (i.e., the engine reached the location where the attack has to be made), the Python engine switches to the attack phase. During this phase, the engine delivers the payload for a given attack. The penetration testing approach for such a phase is to test every entry point on the target page, i.e., using every available input on the page in order to deliver every possible payload in the Instantiation Library. A complete scan of the web applications used as case studies (i.e., WebGoat, DVWA and Gruyere) is not the focus of this paper (and their vulnerabilities are well known anyway); thus, in the implementation of this phase, the Python engine is given the knowledge of the entry points for each of the tested attacks (i.e., the right target field in a form or the partial URL to be used). This choice is not a limitation since the number of entry points can be increased if necessary and this information is only used during the attack phase (i.e., the other phases do not have any knowledge about the exact location of the vulnerabilities and everything is derived from the model and the interaction with the web application). In this way, it is possible to show the effectiveness of the Python engine with a reduced overhead during the attack phase. Of course, with an augmented number of entry points the overall time of the test will increase, but such analysis is not in the scope of this paper.

2.3.4. Check Phase After the delivery of every payload in the previous phase, a check phase starts. If the information contained in the counterexample requires that the check for an attack has to be made in a different location of the web application (i.e., the success of the attack is not verifiable in the received HTTP response), an additional browsing phase is called. Once the engine retrieves the HTTP page where the results of the attack should appear, the engine checks if the attack was successful or not. Two types of checks are implemented in the prototype version of MobSTer: (i) *general checks* and (ii) *payload-related checks*.

General checks include those general conditions of the page that have to be checked for every payload of a given attack (e.g., search for a regular expression in the page, check a given status code, etc.). Payload-related checks include the checks for those attacks where the conditions that have to be checked depend on every single payload. Both *general checks* and *payload-related checks*, for the payloads considered by MobSTer, are included in the Instantiation Library so that the security analyst does not have to manually define them. As an example, for the following payload for a XSS attack

```
alert(String.fromCharCode(88, 83, 83, 65, 116, 116, 65, 99, 107))
```

the string `XSSAttAck` (i.e., the decoded string of the payload) has to be found in the result page in order to have the confidence that the attack was successful. Even though the checking phase

DOI: 10.1002/stvr

implemented in the prototype version of MobSTer is simple, the modularity of the tool allows security analysts to implement and use the test oracle of their choosing.

Taking stock One of the advantages of our concretization phase is that the requests are generated at runtime from both the data acquired through the interaction with the web application and the data derived form the model. This means that we can save and reuse the static variables (i.e., the ones that are received from the server and have to be retransmitted). In its current prototype version, MobSTer is able to handle information that can be displayed and sent with HTML pages. This includes forms, hidden fields and random tokens that are sent by the server and should be sent back for the correct creation of a request. An extension of the Test Execution Engine to handle different technologies (e.g., JavaScript) is left for future work.

3. MODEL INSTANTIATION IN THE ALLOY LANGUAGE

This section shows how the transition system defined in MobSTer by applying the definitions in Section 2.2, can be translated in a model written in the Alloy language.

3.1. The Alloy language

Before showing the translation from the transition system to the Alloy language, we briefly introduce some basic notions of Alloy that are needed to understand the translation and thus the models.

3.1.1. Signatures and Relations A signature (`sig`) is the basic building block of the Alloy language and defines a set of elements. A signature can be specified to have always exactly one element by using the keyword `one`. For example,

```
one sig Obj {};
```

defines a set `Obj` that contains only one element.

A signature can be defined to be `abstract` when one wishes to refine a classification of a set of elements. Each element contained in the abstract signature is constrained to also be contained in the signature that extends the abstract signature itself. An abstract signature can be extended by using the keyword `extend`. For example,

```
abstract sig Color {};

sig Blue extend Color {};

sig Red extend Color {};
```

defines an abstract signature `Color` and two signatures `Blue` and `Red` extending `Color`.

The body of a signature, which is contained within a pair of curly braces, allows one to define fields that declare relations between the set defined by the signature and another set or another relation. To define complex relations, Alloy provides multiplicity constraints and operators on sets. The multiplicity constraints relevant for MobSTer are:

- `x: set e`: meaning `x` is a subset of `e`;
- `x: one e`: meaning `x` is a singleton subset of `e`.

The operators on sets relevant for MobSTer are:

- `X + Y`: the union of sets `X` and `Y`,
- `X & Y`: the intersection of sets `X` and `Y`,
- `X - Y`: the difference of sets `X` and `Y`,
- `R -> S`: the product of two relations.

3.1.2. Facts In Alloy, facts are used to put explicit constraints on the model. During the analysis of a model, any execution trace that violates any facts, will be discarded.

DOI: 10.1002/stvr

3.1.3. Predicates Predicates allow one to specify parameterized constraints that can be used to represent operations. For example,

```
pred name [parameter1:domain1, parameter2:domain2]
     constraint1
     constraint2
     constraint3
```

If the inputs satisfy all of the specified constraints, then the predicate evaluates to true, otherwise it evaluates to false.

3.1.4. Assertion Assertions are assumptions about the model that can be checked using the Alloy Analyzer. For example,

```
assert name-of-assertion
    // list of constraints
```

3.2. Model definitions in Alloy

It is now possible to define how to translate the transition system describing a web application in MobSTer into the Alloy language by means of the definitions in Section 2.2.

3.2.1. Users, data and knowledge The model of a web application in MobSTer can be formalized by applying Definition 1, Definition 2 and Definition 3. The formalization of the model starts from the definition of the user's data structure (Definition 1):

```
abstract sig User {
  field1: one DataType1
  field2: one DataType2
  ...
  initialK: set Data
  gainK: set Data
}
```

Translated in Alloy, the user is an abstract signature that contains a list of fields. Each user's field is also a signature that extends `Data`. The set `Data` represents the generic data type in the model, and the keyword `one` forces each field to contain exactly one element of the specified data type. In addition to the fields specifically related to the web application that is being modeled, the user's signature always defines the fields `initialK` and `gainK`. which are subsets of the set `Data` (denoted by the keyword `set`). Since `Data` is extended by all the other signatures of the user's data, `initialK` and `gainK` can contain any type of information.

Two main data types are needed to model a web application in MobSTer:

- *basic*: defined by declaring a concrete signature that extends the abstract signature `Data`. For example, `sig BasicDataType extends Data`.
- *structured*: defined by declaring (i) an abstract signature A that extends the abstract signature `Data`, (ii) one or more fields in the signature A, and (iii) abstract signatures relative to the fields of the signature A. For example,

```
abstract sig StructDataType {
  field1: one DataType1,
  field2: one DataType2,
  ...
} extends Data
sig abstract DataType1 extends Data
sig abstract DataType2 extends Data
```

With the definition of the user and the data of the web application, it is possible to instantiate the user's fields.

DOI: 10.1002/stvr

- For each *basic* field, the concrete signature (one for each user) is defined as

```
one sig UserADataType1 extends DataType1 {}
one sig UserADataType2 extends DataType2 {}
```

- For each *structured* field, the concrete signature (one for each user) is defined as

```
one sig UserADataType3 extends DataType3{} {
  field1 = UserADataType4
  field2 = UserADataType5
}
```

- The concrete signature of each user and the relation associated to each field are defined as

```
one sig UserA extends User{}
{

  field1 = UserADataType1
  field3 = UserADataType1
  field2 = UserADataType3
  ...
  initialK = UserADataType1 + ...
  gainK = NoData
}
```

3.2.2. State of the transition system A web application's state in MobSTer is defined by the events that the state generates (Definition 4 and 5) and by the security assertions that the state enforces (Definition 6 and 7). An action describes how the fields of the state will change once the action is executed. Actions are translated into Alloy by (i) declaring an abstract signature `Action` and (ii) declaring, for each action in the model, a concrete signature that extends the signature `Action`. For example,

```
abstract Action {}
one sig Action1, Action2 ... extends Action{}
```

A state is modeled as a signature where the events and the security assertions are the fields of the signature. In addition, in the signature of a state, the following fields are always defined:

- `action` to denote what action the state is performing,
- `user` to denote which user is performing the action, and
- `gainK` to denote the information gained by the user perming the action.

```
sig State
{
  action = one Action,
  user = one User,
  gainK: User -> set Data,

  //Assertions
  assertion1: set Data,
  assertion2: set Data,
  ...

  //Events
  event1: set Data,
  event2: set Data,
  ...
}
```

The evolution of a state is modeled as a fact that defines which fields of the state change during the execution of the action:

DOI: 10.1002/stvr

```
fact
{
  action = Action1,
  user = User1,

  //Assertions
  assertion1: Data1,
  assertion2: Data2,
  ...

  //Events
  event1: Data3,
  event2: Data4,
  ...
}
```

4. A CASE STUDY

This section presents the tests performed for the "Stored XSS" and the "String SQL Injection" lessons of WebGoat [33], which is a deliberately insecure web application maintained by the Open Web Application Security Project OWASP [30] and designed to teach web application security lessons. As will be discussed in Section 5, the web applications WebGoat [33], DVWA [18] and Gruyere [27] are used as benchmark since they provide a large set of documented and non-trivial vulnerabilities (i.e., they require non-trivial skills from a security analyst in order to be tested). This makes them the perfect case studies in order to evaluate the vulnerability coverage and effectiveness of MobSTer.

4.1. Model

The model of WebGoat (shown in Figure 4) refers to the lessons where the user can interact through a graphical interface that permits one to: login to a restricted area, display and modify one's profile, display (and for a subset of users, modify) other users' profiles, and search other users' profiles. The tuple that contains the data used in the model is:

$UserData$ =
(
cred,
id,
prof = (name, address),
session,
)

which states that each user has some credentials (modeled as a unique entity), an identifier, a profile (where the only fields needed in the model are the name and the address), and a session with the web application (granted through the login phase).

To define the actions for WebGoat (Table III), we started with the identification of the key functionalities that we wanted to introduce in the model and, afterwards, we modeled each action describing the specific behavior of WebGoat for its functionalities. As an example, to model the $ListId$ action, we chose to make the security analyst introduce the known IDs in the initial knowledge of each user, and show them every time a $ListId$ action is used. This example illustrates how a security analyst can have some control over the possible modeling choices, and that MobSTer is flexible enough to model different scenarios and behaviors.

4.2. WebGoat model in Alloy

The following illustrates how models are defined using Alloy in order to be used during the model-checking phase of the framework (③ in Figure 2).

DOI: 10.1002/stvr

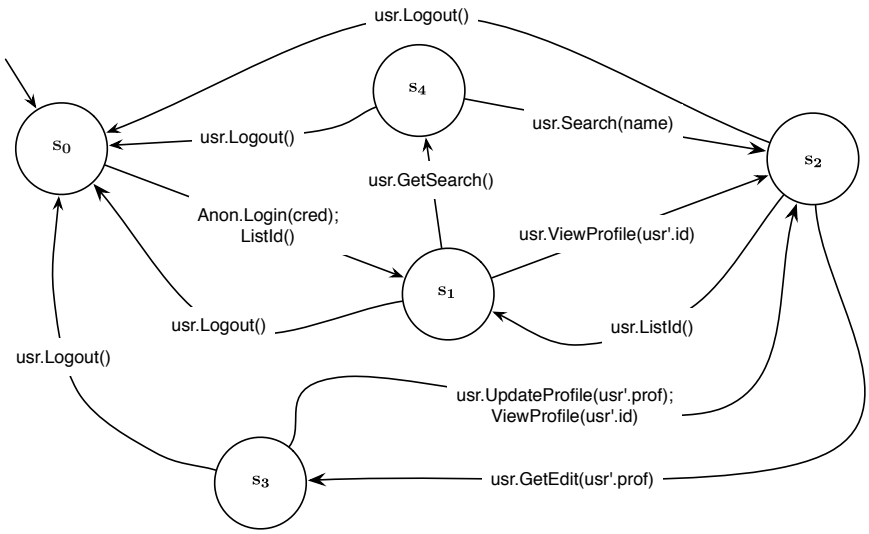

Figure 4. WebGoat case study: high-level model.

Table III. Definition of the actions for the WebGoat case study.

$Login(x,\ x.cred)$
if $(M[Anon, Anon.session] = Grant \wedge M[x, x.cred] = Initial)$
Reset M for x
Del $Grant$ from $M[Anon, Anon.session]$
Add $Grant$ into $M[x, x.session]$
Add $Checked$ into $M[x, x.cred]$
End

$ViewProfile(x,\ y.id)$
if $(M[x, x.session] = Grant \wedge x \neq Anon \wedge M[x, y.id] = ShowDB)$
Reset M for x
Add $ShowDB$ into $M[x, y.profile]$
End

$GetSearch(x)$
if $(M[x, x.session] = Grant \wedge x \neq Anon \wedge (M[x, y.name] = Initial \vee M[x, y.name] = Gained))$
Reset M for x
Add $Edit$ into $M[x, y.name]$
End

$UpdateProfile(x,\ x.prof)$
if $(M[x, x.session] = Grant \wedge x \neq Anon \wedge M[x, x.profile] = Edit)$
Reset M for x
Add $WriteDB$ into $M[x, x.name]$
End

$ListId(x)$
if $(M[x, x.session] = Grant \wedge x \neq Anon \wedge (M[x, y.id] = Initial \vee M[x, y.id] = Gained))$
Reset M for x
Add $ShowDB$ into $M[x, y.id]$
End

$GetEdit(x,\ x.prof)$
if $(M[x, x.session] = Grant \wedge x \neq Anon \wedge M[x, y.profile] = ShowDB)$
Reset M for x
Add $Edit$ into $M[x, x.profile]$
End

$Search(x,\ y.name)$
if $(M[x, x.session] = Grant \wedge x \neq Anon \wedge M[x, y.name] = Edit)$
Reset M for x
Del $Edit$ from $M[x, y.name]$
Add $Checked$ into $M[x, y.name]$
Add $ShowDB$ into $M[x, y.profile]$
End

$Logout(x)$
if $(M[x, x.session] = Grant \wedge x \neq Anon)$
Reset M for x
Del $Grant$ from $M[x, x.session]$
Add $Grant$ into $M[Anon, Anon.session]$
End

4.2.1. Data Used in the Model Section 2.2 defined the data used in the models of web applications, and Section 4.1 defined the ones for WebGoat. In the Alloy models, the set $UserData$ associated

DOI: 10.1002/stvr

Table IV. Definition of the initial state for the WebGoat case study.

	$A.ses$	$T.cred$	$T.id$	$T.name$	$J.cred$	$J.id$	$J.name$
Anon	*Grant*						
Tom		*Init*	*Init*	*Init*			
Jerry			*Init*	*Init*	*Init*	*Init*	*Init*

with the different case studies is defined by assigning an abstract signature to each data. In the high-level model for WebGoat in Figure 4, the abstract signature is defined as:

```
abstract sig Data {}
abstract sig Id, Credential, Name, Addr, UserType,
           Session extends Data {}
```

We instantiate each variable for every user of the web application that we want to model. The signatures for Alice, Bob and the anonymous user considered in the case study are:

```
one sig AliceId, BobId, NoId extends Id {}
one sig AliceCredential, BobCredential,
        NoCredential extends Credential {}
one sig AliceName, BobName, NoName extends Name {}
one sig AliceAddr, BobAddr, NoAddr extends Addr {}
one sig AliceSession, BobSession,
        AnonSession extends Session {}
```

The field *Profile* is composed by the subfields *Name* and *Address* (*Addr* for short), and is defined as an abstract signature for the profile and instantiated for each user:

```
abstract sig Profile extends Data {
  name: one Name,
  address: one Addr   }

one sig BobProfile extends Profile{} {
  name = BobName
  address = BobAddr   }

one sig JerryProfile extends Profile{} {
  name = JerryName
  address = JerryAddr }

one sig NoProfile extends Profile{} {
  name = NoName
  address = NoAddr    }
```

With all the data introduced in the Alloy model as signatures, the definition of the abstract data structure for the users is:

```
abstract sig User{
  profile: one Profile,
  id: one Id,
  name: one Name,
  credential: one Credential,
  session: one Session,
  initialK: set Data,
  gainK: set Data
}
```

and the instantiation for each user with his data is:

DOI: 10.1002/stvr

```
one sig Bob extends User{
  profile = BobProfile
  ID = BobId
  name = BobName
  credential = BobCredential
  session = BobSession
  initialK = credential + BobId +
  AliceId + AliceName + BobName
  gainK = NoData
}
```

A fact is used in to represent how the knowledge of the users can evolve through events by defining how the set $gainK$ evolves between the different states of the model.

```
fact {
  all s: State, s': s.next{
    s'.gainK[s.user] = (s.gainK[s.user] +
    s.showDB + s.showSD + s.showFS ) &&
    (all u : User | (u != s.user) implies s'.gainK[u] = s.gainK[u])
  }
}
```

4.2.2. States As introduced in Definition 8, a state of the transition system is defined by a matrix M such that the row names take values in $UserName$, the column names take values in every element of $Data$ and the resulting cells take values in $\{K_{Source} \cup Event \cup Assertion\}$. A state in the Alloy language is defined as shown in Table V, where the signatures referring to the data are assigned to the elements in K_{Source}, $Event$ and $Assertion$. Since the elements in $UserName$ and $Data$ can vary significantly from one web application to another, and the sets K_{Source}, $Event$ and $Assertion$ are more likely to remain stable (i.e., can slightly change in the long period), the choice of modeling the states in such a way simplifies the modeling activity of the security analyst (i.e., the definition of the states remains unchanged during the testing of different web applications).

Table V shows an example of initial state where `first.X` refers to the first state from which Alloy starts its execution.

A fact defines which actions are usable and thus how to move from a state `s` to its successor `s'`:

```
fact {
  all s: State, s': s.next{
    Login[s,s'] or ListId[s,s'] or GetEdit[s,s'] or
    ViewProfile[s,s'] or UpdateProfile[s,s'] or
    Search[s,s'] or GetSearch[s,s'] or Logout[s,s']
  }
}
```

4.2.3. Actions As introduced in Section 2.2, an action is a functionality of the web application. To make the actions accessible to the Alloy Analyzer, the abstract signature `abstract sig Action {}` is defined and extended with the names of the functionalities ($functionName$) implemented in WebGoat:

```
one sig Login, Logout, ListId, ViewProfile,
        GetEdit, UpdateProfile, GetSearch,
        Search, NoAction extends Action {}
```

To introduce the actions (defined in Section 2.2) in the Alloy model, their definitions have to be translated into Alloy predicates according to the following rules:

- `name`: unchanged;
- `agent`: as a condition on the session granted in `s.granted`;
- `parameters`: as a condition on the knowledge of the user at state `s`;
- `if condition`: as a condition on the knowledge of the user at state s;
- `Add X into M[A,D]`: as an assignment of value in $Data$ (`M[A,D]`) to a variable in AP (`X`);
- `Del X from M[A,D]`: as the assignment of the value `NoData` to the target element in AP (`X`);

DOI: 10.1002/stvr

Table V. States definition and initial state definition for the Alloy models.

```
sig State {
  //User
  user: one User,
  action: one Action,
  //Sec. & Test Info
  grant: set Data,
  checked: set Data,
  AJAX: set Data,
  PageIncluded: set Data,
  echo: set Data,
  exec: set Data,
  //Web applications' Events
  showDB: set Data,
  writeDB: set Data,
  edit: set Data,
  writeSD: set Data,
  showSD: set Data,
  writeFS: set Data,
  showFS: set Data,
  noAttack: set Data,
  gainK: User -> set Data
}
```

```
fact{
  //User
  first.user = Anon
  first.action = NoAction
  //Controls' Predicates
  first.granted = AnonSession
  first.checked = NoData
  first.PageIncluded = NoData
  first.echo = NoData
  first.exec = NoData
  //Web applications' Events
  first.showDB = NoData
  first.writeDB = NoData
  first.edit = NoData
  first.writeSD = NoData
  first.showSD = NoData
  first.writeFS = NoData
  first.showFS = NoData
  first.AJAX = NoData
  first.noAttack = NoData
  (all u:User | first.gainK[u]
  = NoData)
}
```

- `Reset M for A`: as the assignment of the value `NoData` to all the elements of AP (that are not part of other operations) in the matrix.

As an example, the translation of the action *Login* into an Alloy predicate is:

Action Definition	Alloy definition
Login$(x,\ x.cred)$	`pred Login[s, s' : State]`
`if` $M[An, An.sess] = Granted$	`s.granted = AnonSession`
`if` $M[x, x.cred] = Initial$	`u.cred in u.initialK`
`Del` *Granted* `from` $M[An, An.sess]$	`s'.granted = NoData`
`Add` *Granted* `into` $M[x, x.sess]$	`s'.granted = u.session`
`Add` *Checked* `into` $M[x, x.cred]$	`s'.checked = u.cred`
`Reset` M `for` x	Assignment of `NoData` to all remaining elements

4.3. Goal: Stored XSS

In this example, the security analyst is required to execute a Stored XSS attack against the *Street* field on the *EditProfile* page (as *Tom*) and verify that the user *Jerry* is affected by the attack. To find the possible entry points of the web application for stored XSS attacks, we use the goal introduced in Definition 11 (that is translated into an Alloy assertion) from an initial state (defined in Table IV).

4.3.1. Attack trace(s) Alloys's assertions are verified (or falsified) using a check statement, e.g.,

```
check StoredXSS for 10 State, 2 User, 10 Data,
```

which means that the model checker will search for a counterexample that could use the 10 data, bounded to 10 states (i.e., actions) of the web application execution. For what concerns this bound, note that the analysis of the model in MobSTer typically starts from the lowest number of states,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

which is usually two (i.e., the initial state and an action). If no counterexample is found, the web application state bound should be increased by the security analysis as much as she sees fit.

In general, the output of the model-checking phase is a set of counterexamples[10] that violate the goal or a statement "No counterexample found. Assertion may be valid.", which means that the model is secure (under the considered bounds). For this case study, Alloy returns the following counterexample:

```
Trc0=[NoAction, Login, ListId, ViewProfile, GetEdit,
      UpdateProfile, Logout, Login, ListId, ViewProfile]
Sko0=[5, TomProfile, 9, Tom, Jerry]
```

Every trace returned by the Alloy Analyzer starts with "`NoAction`" and ends with the action that permits one to test the vulnerability or check if the attack can be triggered in the web application (as shown in this example).

For every trace (e.g., $\texttt{Trc}_0$) a `Skolem` constant is generated.[11] The Skolem constant is generated from (i) the goal that is checked and (ii) the evolution of the transition system that violates the goal; thus the different parts of a Skolem constant maintain the intended meaning of the associated goal. As an example, $\texttt{Sko}_0$ (generate with the goal `StoredXSS`) states that the web application is attacked at state `5` using `TomProfile` as an entry point and `Tom` as user, and at state `9` the checks about the possibility of triggering the attack are made using agent `Jerry`.

The counterexample also contains the values of the sets K_{Source}, $Event$ and $Assertion$ in the different states. In the Alloy model, the state definition is changed in an orthogonal way with respect to the one presented in Definition 8. This is due to the fact that we want the components of the models to be reusable and thus binding them to the data of the model is not the correct choice (since data can change for different web applications but the three sets K_{Source}, $Event$ and $Assertion$ can be used to model multiple web applications).

As an example, the following code refers to the evolution of `WriteDB` and `ShowDB` through the different states of the transition system:

```
ShowDB  = [,         , [TomId, JerryId], TomProf,...]
CheckDB = [, TomCred,                  , TomId  ,...]
```

Merging the information contained in these sets, it is possible to have a vision of what is happening to the data handled by the web application. In the example, at state 1, the credentials of Tom are checked on the database, at state 2, the IDs of Tom and Jerry are displayed, and at state 3, the ID of Tom is checked and his profile displayed.

4.3.2. Configuration Values Since the data in the model is an abstract representation of the actual data handled by the web application, it is not possible to define a direct relation between them. As discussed in Section 2.3, the security analyst has to define a configuration file (Table II) to fill the gap between abstract and real data. The starting point for the interaction with the web application is stored in the variable `starting_URL` (line `1` in Table II). From this URL, the actions that compose the attack trace are executed sequentially following the concretization methodology presented in Section 2.3 and using the configuration file in Table II.

4.3.3. Instantiation Library As stated before, the Instantiation Library contains information related to the attack; for brevity, we will call such information "payloads". Since different web applications may react differently for each action/payload (e.g., the success of a brute-force attack is not related to the payload used but to the correct execution of the action login), in the Instantiation Library a security analyst can also specify the expected result for each payload or a list of success criteria for an action. The analyst has thus to insert in the Instantiation Library the success criteria that are

[10] MobSTer allows one to automatically perform tests for counterexamples, which means that a large number of counterexamples can be easily managed.

[11] In Alloy, quantified formulas can be reduced to equivalent formulas without the use of quantifiers. This reduction is called Skolemization and is based on the introduction of one or more Skolem constants or functions that capture the constraint of the quantified formula in their values.

DOI: 10.1002/stvr

Table VI. Some of the payloads used during the tests.

(a) Payloads for XSS attacks

```
1  alert(String.fromCharCode(88,83,83,65,116,116,65,99,107))
2  <script>alert(String.fromCharCode(88,83,83));</script>
3  alert("XsSatt");
4  <script>alert("XSSatt");</script>
5  <onmouseover="alert(1)"href="#">readthis!</a>
6  red'onload='alert(1)'onmouseover='alert(2)
```

(b) Payloads for bypassing the login phase

```
1  Administrator'--"
2  root'--
3  ' HAVING 1=1 --"
4  1' or '1'='1"
5  1 or 1=1"
6  101 or 1=1!
```

best suited for the web application. In order to do so, she has to detect on the target web application which information discriminates a successful execution of an action from an unsuccessful one.

The payloads used for this WebGoat lesson are contained in a list of scripts; as an example, Table VI(a) shows a list of some of the payloads used during the test. For every payload, an expected value is saved in order to check if the attack can be triggered, e.g., the first payload should be executed and decoded as "*XSSAttAck*" by the browser.

4.3.4. Test results The tests performed show that MobSTer was able to successfully find and exploit the vulnerabilities in WebGoat. As will be discussed in Section 5, we compare MobSTer with four state of the art tools, showing that MobSTer is the only tool able to find and exploit the vulnerabilities. Having to trigger the attack in a different location from the one where the payload is delivered, the results of the tests are positive only for MobSTer, while the other tools fail to find the attack or test this vulnerability.

The success of MobSTer resides in the attack traces that are generated by the model checker. The fact that the checking phase can be made in a different location from the one used during the attack phase makes MobSTer able to deal with those vulnerabilities that require a complex control on the entry points of the attack and the locations where the attack has to be triggered.

4.4. Goal: Bypass the login phase

The majority of the authentication mechanisms rely on databases for the storage of the credentials associated with users. The credentials provided by a user during the authentication phase are checked with respect to those stored in the database by the web application. In this example, the security analyst is asked to bypass the authentication mechanism without knowing the correct password. In the WebGoat lesson, this is achieved by using a string SQL-Injection[12] that invalidates the password check. In order to search for possible entry points for this attack, the following goal is defined:

Definition 12 (Bypass the login phase)
Let $i \in \mathbb{N}$, $x \in UserName$, and ϱ be an execution fragment. We can check whether there is the possibility of bypassing a login phase if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1 M^1\alpha_2 \ldots \alpha_i M^i$ such that $Checked \in M^i[x, x.cred]$.

4.4.1. Attack trace(s) The `check` statement used during the model-checking phase is:

[12] In WebGoat, SQL-Injection attacks are divided by discriminating on how the inputs are treated by the backend server, i.e., string, numeric and XPATH.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
check BypassLogin for 2 State, 2 User, 6 Data
```

The Alloy Analyzer returns the counterexample:

```
Trc0=[NoAction, Login]
Sko0=[1, Jerry]
Trc1=[NoAction, Login]
Sko1=[1, Tom]
```

4.4.2. Configuration Values Even though the WebGoat lesson used in this example is different from the previous one, the models used in the two examples are the same. Also the Configuration Values are the same as those presented in Section 4.3.2 and Table II.

4.4.3. Instantiation Library The payloads used for this example are contained in a list of general purpose queries for bypassing the login phase; Table VI(b) shows a list of some of the payloads used during the test.

4.4.4. Test results In this example, MobSTer can use payloads in order to bypass the authentication mechanism and gain access to an account without knowing the password of the target user. For what concerns the benchmark tools, only Paros was not able to find the vulnerability, whereas Burp and ZAP did. This is due the fact that Paros does not include the proper payload allowing it to successfully exploit the vulnerability (see Section 5 for more details). It is worth to mention that the versatility of MobSTer permits one to test web application functionalities (e.g., the login) for a variety of attacks with the selection of a different goal during the model-checking phase.

5. EVALUATION

In addition to the case study discussed in detail in the previous section, we have considered three further case studies to assess the strength of MobSTer.

As already stated in Section 1.2, MobSTer provides a hybrid approach that takes advantage of both model-checking techniques and penetration testing guidelines and checklists. More specifically, MobSTer provides an automatic tool that searches for and exploits possible vulnerable entry points without requiring the security analyst to perform manually the analysis (i.e, the search and exploit). The aim of this section is to show the vulnerability coverage and effectiveness provided by MobSTer.

Penetration testers have the option of using many different tools during their penetration testing sessions. However, none of these tools has really the same goal as MobSTer. To compare the relative strengths of MobSTer, we could have considered well-known state-of-the-art tools for exploiting SQL-Injection attacks such as sqlmap [40] or sqlninja [41], but MobSTer is not a tool for finding SQL-Injection payloads. We thus decided to focus on the four security tools that are the closest to what MobSTer aims to achieve: Burp Suite [36] (free version 1.7.23 and Pro version 1.7.23), OWASP Zed Attack Proxy [31] (ZAP, version 2.3.1), Paros [12] (version 3.2.13) and Arachni [2] (framework version 1.5.1).

Arachni is a security scanner framework, whereas Burp Suite, ZAP and Paros are mainly proxy tools used to intercept and analyze the HTTP traffic from and to a web application, but they also provide some basic vulnerability-scanning techniques that we employed for the tests. We are aware of the fact that they are not the most powerful tools for performing vulnerability scanning, but still they are the main general-purpose and free tools currently available. These proxies also provide for the execution of scripts defined by the security analyst that could also be reused. However, we point out that the notion of reusability of scripts in a proxy differs considerably from the notion of reusability that underlies actions. Reusing a script is the result of a decision taken by the penetration tester who already identified a vulnerable point and wants to perform tedious tasks automatically. The reusability of an action is exploited in the modeling phase when the security analyst is creating the model and uses actions for describing the AUT. It is important to highlight this different notion of reusability in order not to mistake an action for an automatic script reusable throughout different penetration testing sessions for performing automatic tasks.

DOI: 10.1002/stvr

5.1. Case studies and their models

In order to perform the tests with MobSTer, we have modeled four case studies: WebGoat, Gruyere [27], Damn Vulnerable Web Application (DVWA [18]) and OnlineShop.

- In the WebGoat case study (Section 4.1), we have used various models:
  - We have used the general model of WebGoat (depicted in Figure 4) in order to test different lessons.
  - We have defined one-action models (i.e., models composed by a single action) for different lessons of WebGoat. This choice reflects the structures of WebGoat (presenting an attack for each lesson even if it is composed only by one functionality). This type of model helps to show the testing capabilities of the framework rather than its model-checking phase.[13]
- Gruyere is a small web application containing multiple security bugs that allows its users to publish snippets of text and store assorted files. The Gruyere model is composed of actions that permit users to add/delete messages (snippets), upload files, modify their profile, and see other user profiles. This variety of functionalities helps in (i) testing different vulnerabilities, and (ii) showing versatility of the framework in testing these vulnerabilities.
- DVWA is a PHP/MySQL web application that suffers from various vulnerabilities. This model follows the division into pages of the real web application (where every page contains a vulnerable entry point for a known vulnerability). From the perspective of the model checker, the selection of the different actions during the model-checking phase is quite simple. The variety of vulnerabilities makes it interesting for proving the testing capabilities of the framework.
- OnlineShop is a realistic case study that we designed to mimic a web application for electronic commerce. This web application has the features that most online shops have, i.e., a catalog of products to be sold, and a basket for each user where items can be added before the purchase. We assume that, to purchase some goods, a user has to (i) finalize the order (i.e., confirm the will of purchasing the items contained in the basket), (ii) enter the payment information and the delivery information, and, in the end, (iii) confirm the order (i.e., the information that he gives during this process). This model is used to derive counterexamples for logic flaws and thus proves that the model-checking phase can be used also in this direction.

5.2. Tool comparison

We have evaluated MobSTer along with four tools normally used by penetration testers: Burp Suite (free version 1.7.23 and Pro version 1.7.23, which is available for a 14 days free trail), ZAP, Paros and Arachni. Burp Suite, ZAP and Paros are mainly proxies that allow the penetration tester to intercept and modify outgoing and incoming requests, whereas Arachni is a web application scanning framework. For performing our comparison we configured the tools in complete automatic mode.

The free version of the Burp Suite does not allow for an automatic scanning and does not have an internal engine able to identify whether an attack has successfully been exploited, so manual search for evidence was required. It does not even provide default payloads to be tested on the web application and thus we had to use Burp by manually providing the payloads used during the test. The result mainly indicates the effectiveness of the provided payloads. Overall, the lack of payloads, and of an automatic engine showing the effectiveness of such payloads, resulted in some effort that we needed to put in order to use Burp for the search of exploits.

[13] During the model-checking of the model composed by only one action, the goals are checked only for the single action and its properties; nonetheless, the result can be easily extended to more complex models.

DOI: 10.1002/stvr

The professional version of the Burp Suite provides an automatic scanner with some payloads already built-in along with an automatic check phase that can be used out of the box.

ZAP and Paros, on the other hand, provide more free functionalities. They have an internal engine that tries to exploit the given URL and provide a final report of the findings. The testing was quite simple: both ZAP and Paros, in automatic mode, ask the user to provide an HTTP request to be used for testing and they automatically understand the parameters used in the request and start the exploitation providing different payloads. Even if the two tools seem to be quite similar, the results are different. ZAP has a better coverage than Paros, which is probably due the fact that (according to its download page[14]) Paros is still being used but its development has been discontinued, whereas ZAP is supported by the OWASP community.

Of the four tools we tested, Arachni is probably the one that is slightly more complicated to use and configure. It comes with a command line interface and a web based interface. The web based interface might sound like a preferred choice, but we noticed that it is not very comfortable to use and thus we finally opted for the command line interface. Out of the box, Arachni supports many different vulnerabilities and provides a quite rich set of payloads to test. It also provides a powerful crawler that is able to navigate the web application, but such a powerful crawler might actually create problems during the scan. A common example is when Arachni, during the crawling, reaches the logout functionality and invalidates the current session causing new requests to be redirected to the login page. This behavior can be controlled by instructing Arachni to exclude specific patterns, which should be specified manually.

We could not test more commercial scanners since the most prominent ones (IBM AppScan, Acunetix, HP WebInspect, Tinfoil, AppSpider, Netsparker) do not provide a trial version suitable for a proper comparison testing. Specifically:

- IBM AppScan[15] and HP WebInspect[16] both provide a trial version that can only be used on a demo domain provided by the company.
- Acunetix[17] similarly allows one to test only a demo website only; moreover, details of the location of the vulnerabilities are not provided in the trial version.
- Tinfoil security[18] does not provide a standalone software that can be tested but only an online service. This characteristic prevented us from testing Tinfoil since, for security reasons, our institutions don't authorize us to have machines remotely accessible on the Internet.
- AppSpider[19] provides a demo version of the software with limited capabilities, thus the result would not be accurate.
- Netsparker demo[20] allows one to scan any website but details of the identified vulnerabilities are omitted, which is not suitable for a proper comparative evaluation.

Note that we performed our comparison to evaluate vulnerability coverage and effectiveness, and did not focus on efficiency since it is quite a tricky aspect to measure as one would need to take into consideration the creation of the model and other factors that are quite elusive to measure (such as expertise of the security analyst using the tool, setup of the tool, manual information gathering and so on). In particular, while it is true that other tools do not require a formal model, it is also true that no tool that aims at performing a deep analysis provides a "point-and-click" user experience. We believe that the effort in generating a formal model (considering also that we provide a database of

[14] `http://sourceforge.net/projects/paros/files/Paros/stats/timeline`.
[15] `https://www.ibm.com/developerworks/downloads/r/appscan/`
[16] `https://saas.hpe.com/en-us/signup/try/webinspect`
[17] `https://www.acunetix.com/vulnerability-scanner/download/`
[18] `https://www.tinfoilsecurity.com/tour`
[19] `https://help.rapid7.com/appspider/content/faqs/demo-keys.html`
[20] The page detailing the limitations of Netsparker is accessible after compiling the download form at `https://www.netsparker.com/web-vulnerability-scanner/download/`

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VII. Results of the tests performed on the case studies. Legenda: ✓ – success of the test, i.e., the test has been able to find the vulnerability, X – failure of the test, ∼ – ineffective test or the test is inapplicable to the case study.

	Case Study	MobSTer	Burp	Burp Pro	ZAP	Paros	Arachni
Access Control Flaws							
Path Based Access Control Scheme	WebGoat	✓	✓	✓	✓	X	✓
Business Layer Access Control	WebGoat	X	∼	∼	∼	X	X
AJAX Security							
DOM-Injection	WebGoat	✓	X	X	∼	X	X
Reflected XSS via AJAX	Gruyere	✓	X	✓	X	X	X
Cross-Site Scripting (XSS)							
Stored XSS	WebGoat	✓	X	X	X	X	X
	Gruyere	✓	∼	X	X	X	X
	Gruyere	✓	∼	X	X	X	X
	DVWA	✓	✓	✓	✓	✓	✓
Reflected XSS	WebGoat	✓	✓	✓	✓	✓	∼
	WebGoat	✓	✓	✓	X	X	∼
	Gruyere	✓	∼	✓	∼	∼	∼
	DVWA	✓	✓	✓	✓	✓	✓
File Upload XSS	Gruyere	✓	∼	∼	∼	∼	∼
	DVWA	✓	∼	∼	∼	∼	
Injection Flaws							
Command-Injection (or Execution)	WebGoat	X	X	X	X	X	X
	DVWA	✓	✓	✓	✓	X	✓
SQL-Injection (string, numeric, XPATH)	WebGoat	✓	✓	✓	✓	X	∼
	WebGoat	✓	✓	✓	✓	X	∼
	WebGoat	✓	✓	✓	X	X	∼
	WebGoat	✓	✓	✓	✓	X	∼
	DVWA	✓	✓	✓	✓	✓	✓
	DVWA	✓	✓	✓	✓	✓	✓
Log Spoofing	WebGoat	✓	✓	✓	X	✓	X
Insecure Configuration							
Forced Browsing	WebGoat	✓	✓	✓	∼	X	X
Path traversal	Gruyere	✓	✓	✓	∼	∼	∼
Miscellaneous							
Password Brute Force	DVWA	✓	✓	✓	✓	∼	X
Logic Flaws							
Missing Checks	OnlineShop	✓	∼	∼	∼	∼	∼
Skip Stages	OnlineShop	✓	∼	∼	∼	∼	∼

reusable actions) is not higher than the effort of learning how to use any of the other state-of-the-art tool and techniques.[21]

[21] Learning how to use modern commercial vulnerability scanners (e.g., Netsparker and Nessus) is a fairly manageable task; the main problem with such scanners resides in the fact that a simple scan is not enough and it always should be followed by a manual penetration test.

DOI: 10.1002/stvr

5.3. Results of the tests performed on the case studies

Table VII shows a tabular representation of the results of the tests that we performed. In the following, we describe in detail the results dividing them by typology of attack (as done in the table).

5.3.1. Access-control flaws Access-control flaws usually violate business-level policies and they are extremely difficult to characterize and identify automatically since those policies are unique for each web application. In our case studies, we defined the access control policies (given by the web application) as goals of the model. The goals check if the user has the correct privileges when accessing an action.

- **Path based access control scheme**: MobSTer was able to access resources that are not referenced by the web application and thus automatize this type of attack. Paros was the only tool not able of solving the WebGoat lesson (i.e., was not able to find the vulnerability), whereas the other tools completed it successfully. Similar results are obtainable with penetration testing using scripts that try to retrieve the resources described in the payloads. However, to manually identify all the possible entry points could be very difficult for complex web applications. The framework allows for the automatic detection of such entry points during the model-checking phase and their subsequent test without the need for a security analyst to manually identify such entry points.

- **Business layer access control**: In this lesson, MobSTer was able to gain access to high-privilege actions with a low-privilege account. The vulnerability was thus tested but a false negative reported (i.e., the vulnerability was exploited but not reported as such). The main reason for MobSTer to be able to exploit this type of vulnerability is that we have introduced in the model the means to reason about roles (as abstract security constraints) and derive traces that are meaningful for these tests. However, the fact that the vulnerability was exploited but not reported, highlights the fact that MobSTer requires a more refined attacks check phase.

  On the other hand, the benchmark tools focus their security evaluation on the analysis of the raw HTTP messages exchanged between the browser and the web application and do not reason about the meaning of the data in these messages.

5.3.2. AJAX security To detect and test this type of flaws, the Alloy goal checks if it is possible to perform an attack (depending on the test) with the use of actions that employ AJAX technologies.

- **DOM-Injection**: MobSTer was able to perform tests for blocked functionalities and to bypass the restrictions coded in the DOM. The other tools failed to identify this attack since they could not perform the request containing the vulnerability. The model that we created for this type of attack provides an example of how different types of functionalities can be modeled in our framework and how a security analyst can leverage information that other security tools cannot see.

- **Reflected XSS via AJAX**: MobSTer was able to find and confirm the attack thanks to its concretization methodology. Specifically, being able to control where the attack has to be launched makes the testing for a distinct vulnerability stronger than a broad search for many vulnerabilities. Only Burp Pro was able to perform the attack thanks to its scanner engine.

5.3.3. Cross-Site Scripting (XSS) The considered XSS flaws are:

- **Stored XSS**: The Alloy goal checks if there exists an action where a data, previously written by a user, is displayed by a (different) user. MobSTer was able to trigger XSS attacks (also on targeted data) in locations different from the ones used to deliver the payload. More specifically:

Prepared using stvrauth.cls
DOI: 10.1002/stvr

  - In the WebGoat and Gruyere[22] case studies, the success of MobSTer resides in the attack traces generated by the model checker. Usually, this type of vulnerability is not found with automatic tools since it requires one to check if a specific payload is executed in a different location where it was injected. MobSTer presents a clear advantage in finding and exploiting this vulnerability since it allows for this possibility.
  - In the DVWA case study, all the tools were able to find the vulnerability. This is due to the fact that the messages delivered through the functionality were directly displayed as the result of the submission and thus the tools were able to directly check the attack.

- **Reflected XSS**:[23] The Alloy goal checks if there exists an action where the user input is displayed back to the same user. MobSTer was able to find the vulnerabilities. Like the other examples, this attack is well known and modern security tools are using mature testing methodologies. For the tests performed with Burp free, all payloads returned the status code 200, so it was impossible to see whether the payloads triggered the actual attack (Burp free does not automatically check if a payload is present in the response of the web application). Burp Pro, on the other hand, provides an automatic checking phase that was able to identify the vulnerability. For the tests performed with Arachni, it should be noted that it was not possible to aim correctly the tool on the vulnerable page since WebGoat loads the content of a page based on the value of some parameters in the HTTP request's body, which Arachni does not allow one to specify. The only other way to make body parameters visible to Arachni is to use its integrated proxy feature. We tried it and although Arachni was now able to test the body parameters, the set of payloads used by Arachni caused WebGoat to generate numerous errors, which resulted in Arachni not being able to find the vulnerability.

- **File upload XSS**: The Alloy goal checks if it is possible to use an action where the content of an uploaded file is displayed back to a user. MobSTer was able to upload a file that allows for the execution of an arbitrary script in the web application. This example shows how it is possible to automatize the search for non-trivial attacks. For the benchmark tools, we do not give a negative evaluation because this type of attack is out of scope for them. The possibility of performing this type of tests with MobSTer is an added value for the security analyst who can automatize the tests without performing them manually (and with a considerable investment of time).

5.3.4. Injection flaws The Alloy goals for the injection flaws check if a user can access an action that was previously injected by an attacker.

- **Command-Injection (or execution)**: The Alloy goal for this vulnerability checks if there exists an action where some data is displayed and retrieved as part of the execution of an injected command.

  - In the case of WebGoat, MobSTer was able to find the vulnerability during the model-checking phase but the TEE could not exploit it. Also Burp (Free and Pro) Paros, Arachni and ZAP failed to find the vulnerability. The main problem in this example was that the standard security policies of the Tomcat server blocked the execution of shell commands.
  - In the case of DVWA, MobSTer, Burp (Free and Pro), ZAP and Arachni were able to successfully complete the tests (i.e., find the vulnerability).

[22] The Gruyere case study gives one the means to test two possible Stored XSS attacks: the first attack leverages the snippet creation, the second leverages the HTML attribute in the user profile. Table VII lists the cases in separate rows.
[23] Two lessons of the WebGoat case study can be used to perform reflected XSS attacks: in the lesson "LAB: Cross-Site Scripting – Stage 5: Reflected XSS", the attacker uses the *Search* action of the model in Figure 4 and Table III in order to attack the web application; in the lesson "Reflected XSS Attacks", a shopping cart is implemented and the attack is performed through the PIN value of a credit card number.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

- **SQL-Injection attacks (string, numeric, XPATH)**: The Alloy goal for this vulnerability checks if, during an action, data used in a query on the database results in additional data displayed by the user interface.

  In the case of the different instances of SQL-Injection attacks, MobSTer was able to perform the tests with success. In these cases, the framework is aligned with Burp (Free and Pro); this is due to the fact that the payloads used for the tests are the same even though a manual check of the success of the tests is required with Burp free. On the other hand, Paros was not able to exploit the vulnerability since it is discontinued and not updated with the appropriate payloads. With the Instantiation Library used in MobSTer, the task of introducing new payloads (or vulnerabilities) is as simple as copying them into a textual file. For Arachni, we had the same problem that occurred with the "Reflected XSS" case studies: the payloads generated by Arachni caused numerous crashes on WebGoat.

- **Log Spoofing**: The Alloy goal for this vulnerability checks if, during an action, data used in a query on the database (or file system) is displayed by the user interface. MobSTer was able to perform a Log Spoofing attack (combined with a XSS attack). The web application implemented in this WebGoat lesson, even though very simple, takes into consideration a vulnerability that is increasingly common in web applications since the login functionality often lacks the proper input sanitization. The fact that ad-hoc payloads can be used with MobSTer (and the related attack discovered) increases the strength of the testing capabilities of the framework itself.

5.3.5. Insecure Configuration The Alloy goal for this type of vulnerability checks if a user can make a direct reference to a file hosted on the server.

- **Forced browsing**: MobSTer was able to perform forced browsing attacks from given locations against the AUT. Since this type of attack mainly depends on the Instantiation Library (i.e., the set of locations to be tested), the completion of the lesson only gives a valid indicator of the effectiveness of the payloads used and the possibility of performing this type of tests.

- **Path traversal**: MobSTer was able to perform a path traversal attack. The results of the tests performed with the tools are not unexpected since the payload to be used is too specific to be available in the tools' lists of directory/files. Burp (Free and Pro) was able to find the attack when the proper name of the file was given.

5.3.6. Other vulnerabilities

- **Password brute forcing**: MobSTer can be used in order to automatize brute-force attacks. The Alloy goal used for this vulnerability is the same as the one defined in Section 4.4. The possibility of defining how an attack has to be checked (i.e., if it is successful or not) makes MobSTer able of bringing the automatization a step further and lowering the discovery of false positive attacks (even if a good knowledge of the web application is required of the security analyst). Burp (Free and Pro) and ZAP were able to exploit the vulnerability when provided with the appropriate payloads. The Alloy goal for this vulnerability simply checks if a user accessed a login action and then we rely on the concretization phase to test the vulnerability. Finally, Arachni does not come with a password brute forcing feature.

5.3.7. Logic flaws Web applications' logic flaws are an important class of defects that are the result of faulty application logic but remain outside the scope of most existing tools (e.g., Burp, ZAP and Paros).

In order to test logic flaws, we have modeled the OnlineShop web application and defined two sets of goals that are used to detect logic flaws during the model-checking phase:

- **Missing checks**: The Alloy goal for this vulnerability checks if a payment of an item is tied to only one order. This example shows how MobSTer can be used to test logic vulnerabilities

DOI: 10.1002/stvr

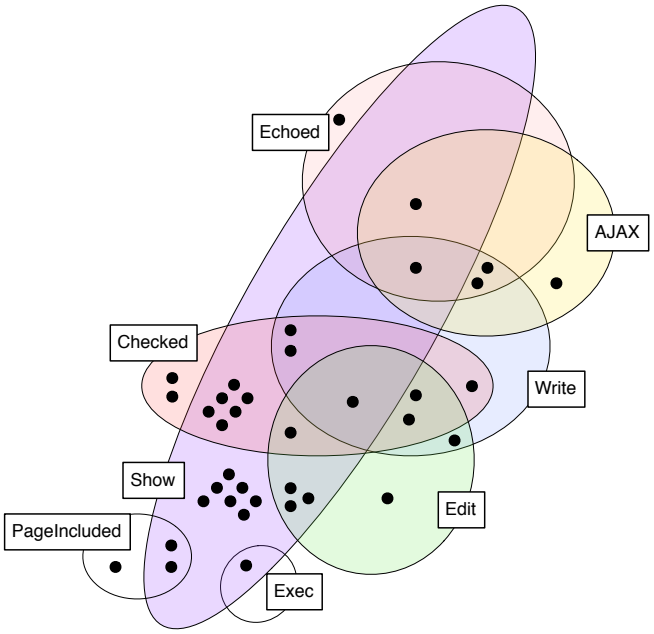

Figure 5. Distribution of the actions (•) with respect to the properties.

where important validations are not implemented by the web application. The definition of the goal reflects the understanding of the security analyst of the process workflow (implemented by the web application) and thus represents the actual flaw, i.e., it contains the assumption on the workflow that the flaw exploits. The benchmark tools focus their tests on well-known vulnerabilities and lack in understanding the "logic" behind a web application.

- **Skip stages**: The Alloy goal for this vulnerability checks if the sequence of actions followed by the user in order to buy an item is in accordance with the web application logic. MobSTer was able to derive tests for multistage mechanisms where different sequences of actions are tested to detect unexpected behaviors of the web application. In this case, the execution fragment itself can be seen as a possible attack trace since the workflow (that the developer wants a user to follow) of such traces is not followed. The main objection to this example is that the modeled web application is fictitious (we devised it but did not deploy it) and thus a proper testing phase could not be performed. This example shows the potential of the model-checking phase in deriving counterexamples for logic flaws; the testing of the framework with similar but real web applications is left for future work.

5.4. On the reusability of actions

As mentioned in Section 2.1, the defined actions can be reused during the tests of different web applications or extended in order to define new ones. Figure 5 shows how the actions (represented with •) used in our case studies are distributed with respect to the properties. To maintain the figure readable, different types of *Write*s are merged into one class (the same simplification is applied to the *Show*s) and the various checks on the sessions (that are used in all the actions) are omitted.

Each action is denoted with • and is placed in one of the subsets created by the intersection of the different properties. An action belonging to a subset means that the property/properties of the subset is/are used in the action definition (we do not discriminate if the property/properties is/are checked in a condition or changed during a transition to the following state). Figure 5 thus shows that the models of the case studies that we considered cover almost all the subsets that can be defined

Prepared using stvrauth.cls
DOI: 10.1002/stvr

with this categorization. This point might sound quite trivial, but we believe that with a database of actions (along with suitable examples) a security analyst has the means to use MobSTer proficiently by first extending the database of actions (changing an action also means that the action has to be placed in another subset of Figure 5) and then defining new actions.

As an example, for the model of the WebGoat lesson "Numeric SQL Injection" (given its simplicity, this lesson is not discussed in this paper), we have defined an action for the selection of a weather station as:

$$
\begin{array}{l}
\mathit{SelectStation}(x,\ x.\mathit{station}) \\
\quad \texttt{if}\ (M[x, x.\mathit{session}] = \mathit{Grant} \wedge x \neq \mathit{Anon}) \\
\qquad \texttt{Reset}\ M\ \texttt{for}\ x \\
\qquad \texttt{Add}\ \mathit{Checked}\ \texttt{into}\ M[x, x.\mathit{station}] \\
\qquad \texttt{Add}\ \mathit{ShowDB}\ \texttt{into}\ M[x, x.\mathit{info}] \\
\texttt{End}
\end{array}
$$

In this simple functionality, the web application searches for a station name in the database and returns the associated information. The above action can be seen as the basis for all the functionalities that share this behavior such as, e.g., a search engine of an e-commerce website. The definition of this action can also be extended; for instance, in the WebGoat model in Section 4.1, the $Search$ action (given in Table III) has the same definition except for (i) the data it refers to and (ii) the fact that it is usable only when the *name* field is editable (i.e., $M[x, y.name] = Edit$). With this case, a security analyst could have defined the $Search$ action starting from the definition of the action $SelectStation$ by changing the name of the action, changing the constraint and the data.

5.5. On the evaluation of MobSTer

The previous subsections give an evaluation (in terms of vulnerability coverage and effectiveness) of MobSTer with respect to four free, non-commercial, state-of-the-art tools. In the following, we discuss the data and impressions we gathered during the definition and the testing of the case studies.

Overall approach: The definition of the case studies confirmed that, in our approach, reusability plays a crucial role. Once the sets K_{Source}, $Event$ and $Assertion$ have been defined, the subsequent definition of actions is straightforward, i.e., the majority of the actions have been defined starting from the ones that we had defined for the previous case studies. What is important to notice is that once the set of actions has been defined, defining a formal model of a web application was easy and straightforward and only required e onto select the actions that describe the web application.

Coverage and effectiveness: As shown in Table VII, MobSTer has a better coverage and effectiveness than the four tools. Two of the key factors for this result can be found in the hybrid approach that MobSTer implements: (i) the model-checking phase automates the identification of possible entry points, whereas (ii) the concretization phase automates the testing of such entry points. With such automatization, MobSTer is better at finding and exploiting vulnerable entry points also when complex behaviors of a web application have to be leveraged in order to attack it.

Scalability and performance: One of the main problems when employing formal analysis techniques is that the computation might be heavy and not terminate. MobSTer is no different since it makes use of the Alloy Analyzer. As a result, the main performance bottle-neck of MobSTer highly depends on two factors: (i) how the formal model of the web application is written and (ii) the implementation of the tool analyzing the formal model. If the formal model is badly written, the computation might get lost in the analysis, while a poor implementation of the formal tool might also result in an endless computation. Since we do not have control on the implementation of they Alloy Analyzer, during the evaluation we tried to keep the models as simple as possible (as highlighted throughout the paper) and as a result the formal analysis finishes in the term of seconds. We believe that reusable actions will also give the modeler the opportunity to create models in a more structured and compact way so as to make the analysis feasible. Exploring performances in more complex scenarios is an interesting challenge that MobSTer will undertake. Given the positive evaluation that we give in this paper, we believe that applying MobSTer to complex scenarios will require some tuning in the source code (see "Possible refinements" below) but eventually give positive results.

DOI: 10.1002/stvr

Limitations: The definition of new goals requires a basic understanding of logic but a coherent definition of the sets K_{Source}, $Event$ and $Assertion$ is essential. Without a consistent definition of these sets since the beginning of the modeling phase, a security analyst might encounter some problems with the continuous changes in the abstract definitions (i.e., the features that the security analyst wants to convey about the web application she is modeling) and the actual model she is defining.

Possible refinements: MobSTer is a prototype implementation that shows the applicability of the approach proposed in this paper, but is not yet mature enough to be used as an industrial tool. We envision that this will require a number of refinements ranging from a stronger modularity of the source code, which will allow for a better maintainability, to an improved *attack phase*, enabled by extending the instantiation library with further payloads.

Taking stock: We believe that the evaluation of MobSTer has concluded with flying colors and that our tool is ready to take the next, more mature step: with MobSTer available it will be possible to apply it to "industrial-strength" case studies, where the tool will help developers, users and analysts of web applications in the identification of previously unknown vulnerabilities.

6. RELATED WORK

We have already mentioned a number of works in the area of penetration testing for web applications ([13, 14, 29, 34, 32, 39, 42]). However, those works are not directly comparable with MobSTer since they are far from being formal and only provide a series of guidelines a security analyst should follow. Furthermore, as stated by Bau et al. [5] and Doupé et al. [17], vulnerability scanners are effective tools for detecting "historical", well-known vulnerabilities (e.g., textbook XSS and SQL-Injection attacks) but perform poorly in detecting more sophisticated vulnerabilities that require a deeper understanding of workflows and dataflows of web applications. It is also interesting to note that during penetration testing performed in industry sometimes scanners cannot be used since clients are not comfortable with the possible disruption of data that can derive from the unattended and uncontrollable requests that such scanners generate.

Model-based testing approaches, on the other hand, are more closely related to MobSTer and thus call for a better comparison; we now focus on the most relevant works.

Calvi and Viganò [11] describe an approach called "Chained Attack" that takes HTTP requests in input, generates a model, and searches for sequences of attacks on the model by exploiting model-checking techniques that implement the Dolev-Yao [16] intruder. The approach is similar to ours, but the result they present is quite limited. Even though the approach aims to be used by security analysts that do not have a strong background in formal methods, quite a lot is required of them for the concretization phase. Security analysts have to write a configuration file containing information such as header names and keys appearing in raw HTTP messages, semantics of HTTP responses and conditions representing attacks, whereas our approach simplifies the definition of the models and provides more automation for the concretization phase. More specifically, their concretization phase is not as mature as ours since they provide only information about the payloads that need to be used, whereas MobSTer provides for a more solid testing engine with its Instantiation Library. Furthermore, "Chained Attack" does not consider many vulnerabilities that we have included in our analysis, making MobSTer able to cover a wider range of vulnerabilities as we consider vulnerabilities like password brute forcing, log spoofing, logic flaws and many others that "Chained Attack" does not include.

The methodology proposed by Armando et al. [3, 4] is focused on binding the specification of security protocols to actual implementations. The methodology starts with the definition of an abstract model (written using a role-based language) of the HTTP messages composing the security protocol. A model checker is then used in order to derive a counterexample violating some given security properties. The abstract messages, contained in the counterexample, are then mapped to concrete messages used to test the web application. The results are particularly promising but not directly comparable to ours, since our framework is at a different level of abstraction. Indeed, the

DOI: 10.1002/stvr

models we are dealing with are not protocol-dependent (HTTP messages) but application-dependent (actions).

The work by Büchler [8], Büchler et al. [9, 10] present an approach for model-based security testing of web applications closely related to mutation testing: they start from a secure model and use mutation operators to automatically introduce vulnerabilities in the model. Some of the major differences with our approach reside in modeling the interaction between user and web application (our approach takes into consideration the user interaction only when we test the counterexamples), the use of mutations and the use of an attacker in the model-checking phase. On the other hand, the approach by Büchler [8], Büchler et al. [9, 10] is more mature than ours and allows for automatic test-case generation via mutants.

The work by Akhawe et al. [1] proposes a methodology that relies on the Alloy Analyzer for the analysis of several sample web mechanisms and web applications. The authors employ threat models such as a malicious attacker controlling a website or even a portion of the network. They have defined three different intruder models that should find web attacks, whereas we only need to describe the behavior of the web application. Furthermore, this methodology is at a different level of abstraction than our framework since they mainly model network infrastructures and protocols rather than web applications.

The work by Lebeau et al. [28] describes an approach for the identification of vulnerabilities based on the formalization of vulnerability test patterns into test purposes. They define both the behavior of the web application and the test purpose, and use model-checking techniques for the generation of abstract test cases. The idea is similar to our approach, but our framework allows for a wider coverage of vulnerabilities, whereas Lebeau et al. [28] considers the main vulnerabilities that are provided in terms of test patterns from CWE (i.e., Blind and Not Blind SQL-Injection, Reflected and Stored XSS) and that need to be translated in order to be tested. The coverage of MobSTer is wider as it considers also more complex attacks that exploit logical flaws. Vernotte et al. [44] extend their work ([28]) with risk assessment in order to select relevant aspects and vulnerabilities for the generation of models. The same differences with respect to our work that we pointed out for Lebeau et al. [28] apply for Vernotte et al. [44].

The work by Blome et al. [7] proposes a model-based vulnerability testing approach where attacker models are used to test a given web application (or functionality). In this approach, the payloads and the behavior of the attacker are separated. Attacker models can be seen as an instantiation of the security goals that are defined in this paper, although the level of abstraction of the security goals described in this paper is higher (in the sense that the attacker models are specific to a scenario) than the one proposed in [7]. The main difference with respect to this approach is that we base our analysis on the models of web applications rather than the procedures for testing them; this results in not needing to change such procedures if a (slightly) different web application has to be tested.

Another formal approach that uses a different attacker model is proposed by Rocchetto et al. [37] where they use the formal language ASLan++ [45] and model-checking techniques that implement the Dolev-Yao [16] intruder model for modeling a web application vulnerable to CSRF but they do not consider other vulnerabilities.

Jürjens [26] proposes a model-based security testing methodology for the systematic generation of test-sequences of security-critical system. The methodology, supported by the UMLsec tool, is applied to test the security aspects of the Common Electronic Purse Specifications (CEPS). During the modeling process (i) the protocol and its participants are specified, (ii) a property to be tested is defined, and (iii) a threat model is included in the system specification (e.g., public channels are vulnerable). Test case specifications are formalized along with the model. The methodology will then generate test sequences, satisfying the test case specification, which correspond to executions when the system is under attack. Closely related to Jürjens [26], Fourneret et al. [21] present an approach for the verification of smart-card specific security properties that combine model-based testing with UMLsec security verification techniques. They present an extension of UMLsec stereotypes (UMLsec profiles of rules that formalize security requirements) for security relevant properties of smart-cards to verify the model and drive the test generation dedicated to security. The

Prepared using stvrauth.cls
DOI: 10.1002/stvr

approach first verifies that the model is consistent with respect to the considered security property and then transforms the UMLsec stereotypes to test schemas that exercise the system to validate that it behaves as predicted by the model.

With regards to our approach, the main advantage of the methodologies by Jürjens [26] and by Fourneret et al. [21] is the close relation between the model of the system and its implementation. This allow them to better verify the correctness of the model and generate test sequences that closely represent the system execution. Nevertheless, the security properties that they consider are tailored around the case studies that they present. The security goals that we present in the paper are web-application agnostic (i.e., they are independent of the specific web application under testing) and suitable for different case studies since they are not directly tied to actions of the web application but to the data that the actions handle.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented MobSTer, a framework that uses model-checking techniques to automate the testing of web applications without missing important checks. Web application models are created by taking into consideration different aspects of web applications: (i) their functionalities (that are modeled as actions), (ii) the data used (along with information about data storage and management), and (iii) a security goal specifying the vulnerability to be tested.

As illustrated by the case studies, the use of actions has a positive impact on all the phases of the framework that we propose, resulting, we believe, in a quite simple and flexible methodology for testing the security of web applications.

One of the strengths of our approach is the reusability of actions and of the sets K_{Source}, $Event$ and $Assertion$. The expertise required to populate the Instantiation Library is automatically "reused". If this strength may sound trivial for the more skilled penetration tester, we believe that the adoption of our framework by a testing group can result in a non-indifferent improvement for the testing capability of the whole group. The analyst can collect her expertise into MobSTer and reuse it during future tests on possibly different web applications, which may be carried out by her or by members of the testing group of the analyst's organization, if any. New attack techniques and payloads can be inserted into the framework without changing (or compromising) the testing methodology.

Regarding the different types of knowledge, we aim to extend the Test Execution Engine with procedures to guess the labeled data from the information that is available at that state of the execution. As an example, three possible approaches for the implementation of a procedure for the creation of a list of guessed inputs could be: (i) check the data available in a particular moment in the Test Execution Engine variables and fuzz them, (ii) employ a regular expression that uses the expected input (i.e., char, int, etc.) and the available data, and (iii) create a set of dictionaries of wordlists for each expected input.

Regarding the security goals, we are aware that a basic knowledge of logic is required in order to write them, but their reusability compensates the efforts put in their definition. As stated before, the security goals are a high-level representation of vulnerabilities, thus, a security analyst can use the same goal in models of different web applications without any problems.

To summarize, we believe that modeling web applications using actions, rather than using messages representing the underlying protocol, has a lot of potential but further work is, of course, still needed. In particular, we are currently working on defining a database of actions with the related data and properties. The idea is to investigate how the granularity of these sets (i.e., the different levels of abstraction we can have in our models) changes the testing effectiveness of our framework.

In Section 5, we have already discussed the wide coverage provided by MobSTer. The tests we performed on the benchmark case studies clearly show an improvement with respect to the state-of-the-art solutions that penetration testers are currently using. We plan to further extend the set of case studies by considering different industrial-strength web applications and performing (or help the developers, users and analysts perform) a security analysis with MobSTer in order to find previously unknown vulnerabilities. This will allow us to increment the number of actions that can

Prepared using stvrauth.cls
DOI: 10.1002/stvr

be used during the modeling phase and, afterwards, to better assess the capabilities of MobSTer against more complex web applications.

Finally, we have been implementing a graphical interface for the security analyst and integrating it in the framework. The idea is to create an interface from which an analyst can select actions and specify their dependencies, and use this information to create the skeleton of the model, which will provide a stepping stone for the full automation of the workflow.

ACKNOWLEDGEMENTS

This paper is largely based on Michele Peroli's PhD Thesis [35] and was partially supported by the EU FP7 Project no. 257876, "SPaCIoS: Secure Provision and Consumption in the Internet of Services" (www.spacios.eu) and the PRIN 2010-11 project "Security Horizons". Much of this work was carried out when Luca Viganò was at the Università di Verona. We thank Matthias Büchler, Alberto Calvi, Martín Ochoa, Johan Oudinet, Alexander Pretschner, Marco Rocchetto and Marco Volpe for useful comments.

REFERENCES

[1] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *23rd IEEE Computer Security Foundations Symposium, CSF 2010*, pages 290–304. IEEE Computer Society, 2010. doi: 10.1109/CSF.2010.27. URL http://doi.ieeecomputersociety.org/10.1109/CSF.2010.27.

[2] Arachni scanner. http://www.arachni-scanner.com/.

[3] Alessandro Armando, Roberto Carbone, Luca Compagna, Keqin Li, and Giancarlo Pellegrino. Model-Checking Driven Security Testing of Web-Based Applications. In *3rd International Conference on Software Testing, Verification and Validation, ICST 2010*, pages 361–370. IEEE Computer Society, 2010. doi: 10.1109/ICSTW.2010.54. URL http://dx.doi.org/10.1109/ICSTW.2010.54.

[4] Alessandro Armando, Giancarlo Pellegrino, Roberto Carbone, Alessio Merlo, and Davide Balzarotti. From model-checking to automated testing of security protocols: Bridging the gap. In *Tests and Proofs - 6th International Conference, TAP 2012*, pages 3–18. Springer, 2012. doi: 10.1007/978-3-642-30473-6_3. URL http://dx.doi.org/10.1007/978-3-642-30473-6_3.

[5] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345. IEEE Computer Society, 2010.

[6] Robert V. Binder, Bruno Legeard, and Anne Kramer. Model-based Testing: Where Does It Stand? *Queue*, 13(1):40–48, December 2014.

[7] Abian Blome, Martín Ochoa, Keqin Li, Michele Peroli, and Mohammad Torabi Dashti. VERA: A flexible model-based vulnerability testing tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013*, pages 471–478, 2013. doi: 10.1109/ICST.2013.65. URL http://dx.doi.org/10.1109/ICST.2013.65.

[8] Matthias Büchler. *Semi-Automatic Security Testing of Web Applications with Fault Models and Properties*. PhD thesis, Technical University Munich, 2015.

[9] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Semi-Automatic Security Testing of Web Applications from a Secure Model. In *6th International Conference on Software Security and Reliability, SERE 2012*, pages 253–262, 2012. doi: 10.1109/SERE.2012.38. URL http://dx.doi.org/10.1109/SERE.2012.38.

[10] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. SPaCiTE – Web Application Testing Engine. In *5th IEEE International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 858–859, 2012. doi: 10.1109/ICST.2012.187. URL http://dx.doi.org/10.1109/ICST.2012.187.

[11] Alberto Calvi and Luca Viganò. An Automated Approach for Testing the Security of Web Applications Against Chained Attacks. In *31st ACM/SIGAPP Symposium on Applied Computing*. ACM Press, 2016. doi: 10.1145/1698750.1698754.

[12] Chinotec Technologies Company. Paros – web application security assessment. http://www.parosproxy.org/.

[13] CVE. Common Vulnerabilities and Exposures. https://cve.mitre.org/index.html, 2016.

[14] CWE. Common Weakness Enumeration. https://cwe.mitre.org/, 2016.

[15] Arilo Claudio Dias Neto and Guilherme Horta Travassos. A picture from the model-based testing area: Concepts, techniques, and challenges. *Advances in Computers*, 80:45–120, 2010.

[16] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–208, 1983. ISSN 0018-9448. doi: 10.1109/TIT.1983.1056650.

[17] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *DIMVA*, LNCS 6201, pages 111–131. Springer, 2010.

[18] DVWA team. Damn Vulnerable Web App (DVWA). http://www.dvwa.co.uk, 2015.

[19] Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler, and Alexander Pretschner. Model-based security testing: A taxonomy and systematic classification. *Software Testing, Verification and Reliability*, 2015. doi: 10.1002/stvr.1580. URL http://dx.doi.org/10.1002/stvr.1580.

[20] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Security Testing: A Survey. *Advances in Computers*, 101:1–51, 2016.

[21] Elizabeta Fourneret, Martín Ochoa, Fabrice Bouquet, Julien Botella, Jan Jürjens, and Parvaneh Yousefi. Model-Based Security Verification and Testing for Smart-cards. In *Sixth International Conference on Availability, Reliability and Security, ARES*, pages 272–279, 2011.

[22] Shashank Gupta and Brij Bhooshan Gupta. Cross-Site Scripting (XSS) attacks and defense mechanisms: Classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 2015.

[23] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Commun. ACM*, 19(8):461–471, 1976. doi: 10.1145/360303.360333. URL http://doi.acm.org/10.1145/360303.360333.

[24] Daniel Jackson. Alloy: A language & tool for relational models – Documentation. http://alloy.mit.edu/alloy/documentation.html.

[25] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012. ISBN 0262017156, 9780262017152.

[26] Jan Jürjens. Model-based Security Testing Using UMLsec. *Electronic Notes in Theoretical Computer Science*, 220(1):93 – 104, 2008.

[27] Bruce Leban, Mugdha Bendre, and Parisa Tabriz. Gruyere: Web Application Exploits and Defenses. https://google-gruyere.appspot.com/, 2015.

[28] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-Based Vulnerability Testing for Web Applications. In *SECTEST'13*, pages 445–452. IEEE CS Press, 2013.

[29] NVD. National Vulnerability Database. https://nvd.nist.gov/home.cfm, 2016.

[30] OWASP. Open Web Application Security Project. https://www.owasp.org.

[31] OWASP. Zed Attack Proxy Project (ZAP). https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

[32] OWASP. Web Application Security Testing Cheat Sheet. https://www.owasp.org/index.php/Web_Application_Security_Testing_Cheat_Sheet.

[33] OWASP. WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2011.

[34] OWASP. Testing Guide v 4.0, 2015.

[35] Michele Peroli. *A Model-Based Security Testing Approach for Web Applications*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Verona, Verona, Italy, 2015.

[36] PortSwigger Ltd. Burp suite. http://portswigger.net/burp/.

[37] Marco Rocchetto, Martín Ochoa, and Mohammad Torabi Dashti. Model-Based Detection of CSRF. In *IFIP SEC*, pages 30–43. Springer, 2014.

[38] Amirmohammad Sadeghian, Mazdak Zamani, and Shahidan M. Abdullah. A Taxonomy of SQL Injection Attacks. *ICICM*, pages 269–273, 2014.

[39] SANS. Securing Web Application Technologies [SWAT] Checklist. https://software-security.sans.org/resources/swat.

[40] sqlmapproject. sqlmap: Automatic slq injection and database takeover tool, 2013. http://sqlmap.org.

[41] sqlninja. sqlninja: A SQL Server injection & takeover tool, 2013. http://sqlninja.sourceforge.net.

[42] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition*. John Wiley & Sons, Inc., 2011. ISBN 978-1-118-02647-2.

[43] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012. doi: 10.1002/stvr.456. URL http://dx.doi.org/10.1002/stvr.456.

[44] Alexandre Vernotte, Cornel Botea, Bruno Legeard, Arthur Molnar, and Fabien Peureux. Risk-Driven Vulnerability Testing: Results from eHealth Experiments Using Patterns and Model-Based Approach. In *RISK 2015*, pages 93–109. Springer, 2015.

[45] David von Oheimb and Sebastian Mödersheim. ASLan++ — a formal security specification language for distributed systems. In *FMCO*, LNCS 6957, pages 1–22. Springer, 2010.

DOI: 10.1002/stvr